USENIX Association

Proceedings of the 3rd Virtual Machine Research & Technology Symposium (VM '04)

May 6–7, 2004
San Jose, California, USA

VM '04: 3rd Virtual Machine Research & Technology Symposium

May 6–7, 2004
San Jose, CA, USA

Index of Authors

Message from the Program Chair

Since the introduction of Java and the Java Virtual Machine in the early '90s, virtual machine technology has become an increasingly important part of the software infrastructure. Today, virtual machines are part of computer systems ranging from small-scale embedded devices to large-scale computing grids. The 3rd Virtual Machine Research and Technology Symposium (VM '04) provides a venue for presenting top-quality research on virtual machines and for the interaction between academic and corporate researchers in this area.

VM '04 builds on the success of the two preceding Java Virtual Machine Research symposia, held in 2001 and 2002. This year, the scope of the symposium expands to encompass other types of virtual machines, such as Microsoft's .NET, and low-level virtual machines/environments.

We received 40 submissions from all over the world. The program committee selected 14 papers that represent some of the best work in the area of virtual machines, on topics that include virtual machine architecture and performance, low-level virtualization, dynamic optimization, and virtual grids. The papers come from all regions of the world, including Europe, Japan, and the Americas, and represent both academic and corporate research.

The symposium's program features two Keynote Addresses by Mendel Rosenblum, Associate Professor of Computer Science at Stanford University, and by Miguel de Icaza, Co-founder and CTO of Ximian. The program also features Work-in-Progress Reports and Birds-of-a-Feather Sessions.

I am grateful to many who made this conference a reality: to the authors who submitted the results of their dedicated work; to the program committee members and external reviewers who spent many hours of their valuable time evaluating the submissions; to our two keynote speakers for taking the time and making the effort to share with the symposium's attendees their visions and insights; and to all the USENIX staff members for their wonderful help in putting the conference together. It has been a pleasure and a privilege to work with each and every one.

Finally, I am very thankful to Microsoft Research for their generous support of the conference in the form of student stipends.

Tarek S. Abdelrahman, ***University of Toronto***
Program Chair

A Virtual Machine Generator for Heterogeneous Smart Spaces

Doug Palmer
CSIRO ICT Centre
`Doug.Palmer@csiro.au`

Abstract

Heterogenous smart spaces are networks of communicating, embedded resources and general-purpose computers that have a wide spread of power and capabilities. Devices can range from having less than a kilobyte of RAM to many megabytes.

Virtual machine techniques can be used to control some of the inherent complexity of a heterogeneous smart space by providing a common runtime environment. However, a suitably rich, single virtual machine implementation is unlikely to be able to operate in all environments.

By using a virtual machine generator and allowing virtual machines to be subsetted, it is possible to provide numerous virtual machines, each tailored to the capabilities of a class of resources.

1 Introduction

A heterogeneous smart space, such as the SmartLands[18] smart space contains many different sensors and controllers, each with their own set of capabilities and, in particular, computing power. Individual devices can range in power and size from a Berkeley Mote (128Kb flash memory, 4Kb SRAM)[4] to a PDA (64Mb RAM)[16]. The smart space, as a whole, can also have access to general-purpose computing resources[21].

A contrast to a heterogeneous smart space is the sort of homogeneous smart space as the Ageless Aerospace Vehicle skin[14], or a Motes network, where the computing resources available tend towards uniformity.

Heterogeneous smart spaces can be expected to appear whenever longevity and cost are overriding issues; in a farm or building, for example. There are a number of factors driving heterogeneity in these environments:

- Pre-existing resources may be built into the smart spaces environment — sensors in the fabric of a building, for example — and difficult to replace or upgrade. A sensor and its associated processing element may be expected to last for the lifetime of the smart space, leading to a 20- or 30-year gap between the oldest elements and the latest introductions, with an associated disparity in performance.
- Resources are introduced into the smart space for a variety of purposes — a temperature sensor and an automatic feeding gate, for example — and may be selected for reasons other than compatibility.
- Those resources that can be upgraded will, most likely, be upgraded piecemeal, when funds and suitable products are available.
- The purpose of the smart space environment may change. For example, a warehouse may be sold and renovated as residence.

The environments which generate heterogeneous smart spaces also tend to generate a plethora of distinct applications, all competing for resources. On a farm, for example, the stock protection, environment monitoring and irrigation systems may all want to use a single temperature sensor for a variety of purposes. New applications may be added, and old ones removed, in an ad hoc manner. These applications will tend to be of ordinary commercial quality, rather than safety-critical quality and will often fail or go awry; the smart space as a whole will need to be protected from rogue applications.

A feature of heterogeneous smart spaces is that common applications — building heating, stock protection, active maps, etc. — need to be deployed into individual, complex smart spaces. To allow smart spaces to be useful at a common, commercial level, some mechanism for automated customisation and deployment is needed. There are two strands to automated customisation and deployment: at the top level, a declarative service description language model is needed, to allow applications to abstract the resources needed to perform a task[17]; at the bottom level, some sort of mechanism is needed to help control the complexity inherent in an ad hoc collection of resources with competing applications.

Figure 1 shows an example top-level deployment onto a field smart space. The smart space consists of some low-level soil moisture sensors with minimal processing power and range, some intermediate-level fence-post processors and a general-purpose monitoring and management facility. The moisture sensors have been "sown" into the field over several years. Each sowing uses whatever agricultural sensor packages are most economical at the time, leading to a mixture of architectures and processing platforms. The sensors form an ad-hoc network with each sensor connecting to any near neighbours.

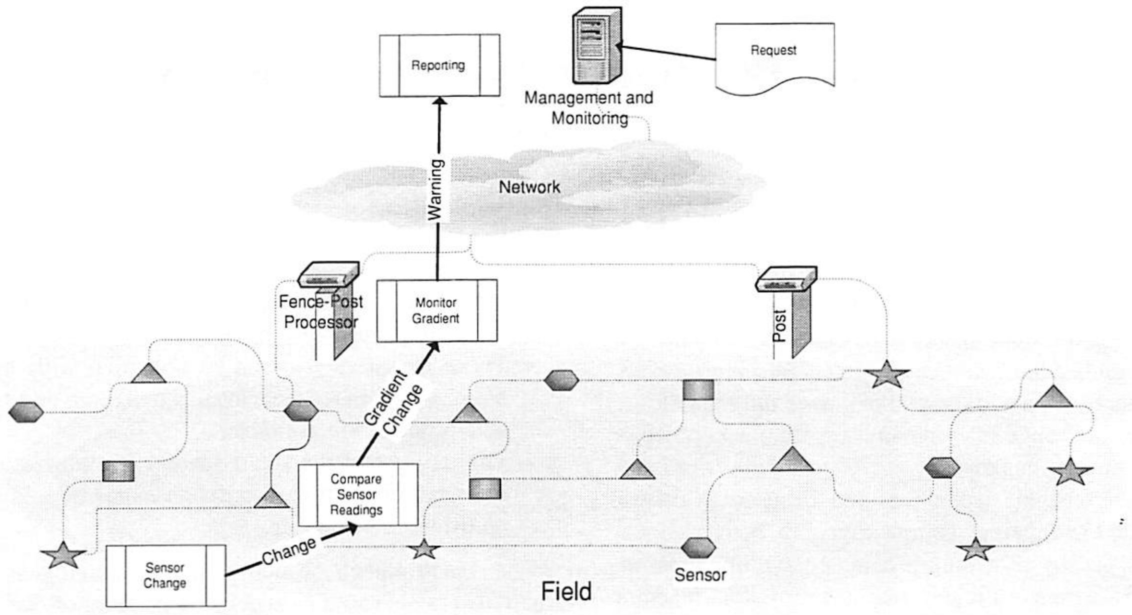


Figure 1: Example Smart Space Deployment

At the top level is a declarative request made by an application to monitor the moisture gradient of the field and raise an alarm if the gradient fluctuates outside acceptable bounds. This request must be mapped onto the smart space by the smart space itself, since the smart space is aware of the resources available, their capabilities and their properties. In the example smart space, sensors report any significant changes in moisture content to neighbours, which then compare the changes with their own readings. If a fluctuation is detected, the event is reported to a nearby fence-post processor, which collates reports in a local area and notifies the monitor of any significant changes. The deployment shown in Figure 1 only shows one instance of each routine for clarity. Each sensor is running both the sensor monitoring and gradient change detection routines.

A consequence of the request is that essentially identical programs need to be run on a wide range of hardware platforms, corresponding to the range of sensors that have been distributed in the field. A bottom-level system that allows a separation between program and implementation would help control the complexity inherent in a deployment across multiple resources. The main requirements for such a bottom-level system can be summarised as follows:

heterogeneity Multiple source representations and multiple machine architectures need to be accommodated. A wide range of computing power and space needs to be supported.

parsimony The system needs to be able to fit into the very limited resources available on some smart spaces environments.

economy Power consumption and network traffic need to be kept to a minimum.

security Hostile or buggy code should have minimal impact.

concurrency Multiple applications may need to run independently on a single resource. A single application may not hog all the resources available.

The advantages of a common language runtime have long been recognised when working with many machine architectures and many languages[15]. A virtual machine allows a *safe* common language runtime to be implemented, with the virtual machine preventing overflows and illegal, unmediated access to resources such as sensors, processor time and memory not allocated to the program being run.

However, some of the computing resources in a smart space are not large enough to handle dynamic strings, let alone something as sophisticated as a full object-oriented environment. There is also a considerable difference in the sophistication required across the range of resources. In the soil moisture monitoring example, there is a considerable difference between the simple monitoring functions performed by a sensor and the more complex array processing required in the fence-post processor, where "significance" is determined.

The approach taken here is to make use of the communications inherent in a smart space. Small resources can use a subset of the full virtual machine, perhaps only capable of simple integer arithmetic. More complex processing can occur on larger resources, capa-

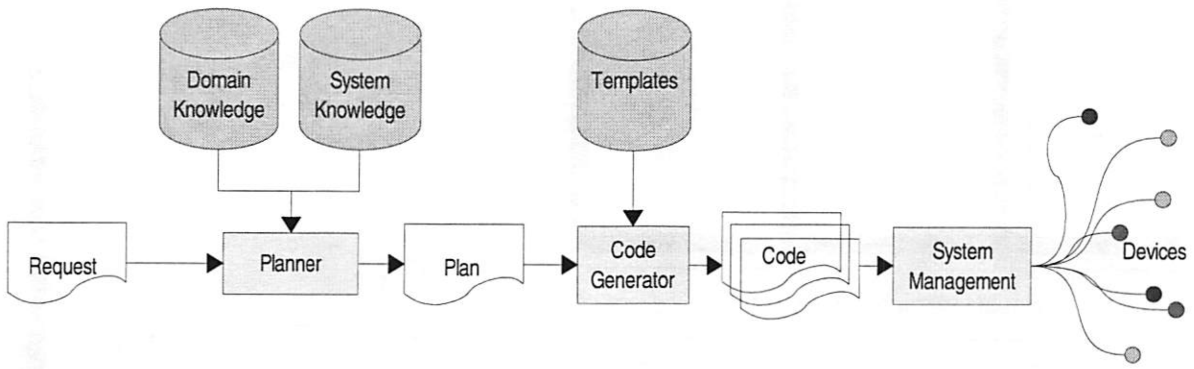


Figure 2: Example Smart Space Code Deployment

ble of more sophisticated processing and memory management. Large scale data and system management can be handled by general-purpose computers[21] or by exploiting the emergent properties of multi-agent systems[14]. An application can be partitioned into fragments of code that can be distributed throughout a smart space, with the low-capability resources offloading sophisticated processing onto their more powerful brethren.

To allow specialised virtual machine subsets, a virtual machine generator is used. An abstract virtual machine specification, along with a description of the subset needed for a particular resource, is fed into the generator. The generator then constructs source code (in C or Java) for a virtual machine that implements the specification. This virtual machine can then be compiled, linked with a resource-specific kernel and loaded into the resource. Application-specific code can be loaded into the running virtual machine across a communications network as components[20].

A sample deployment architecture is shown in Figure 2. Each device has a customised virtual machine, with knowledge about the capabilities of that virtual machine kept in a system knowledge database. A high-level request is given to a planner. The planner uses knowledge about the structure of the smart space and the domain of the request to build a plan: a set of small components (a few subroutines in size) in an intermediate language such as Forth or a subset of C. The plan reflects the known capabilities of the devices and the connections between the devices. Each part of the plan is compiled into code by a templating code generator, which selects code generation templates based on the capabilities of the target virtual machine. The code can then be distributed to the target devices.

1.1 Related Work

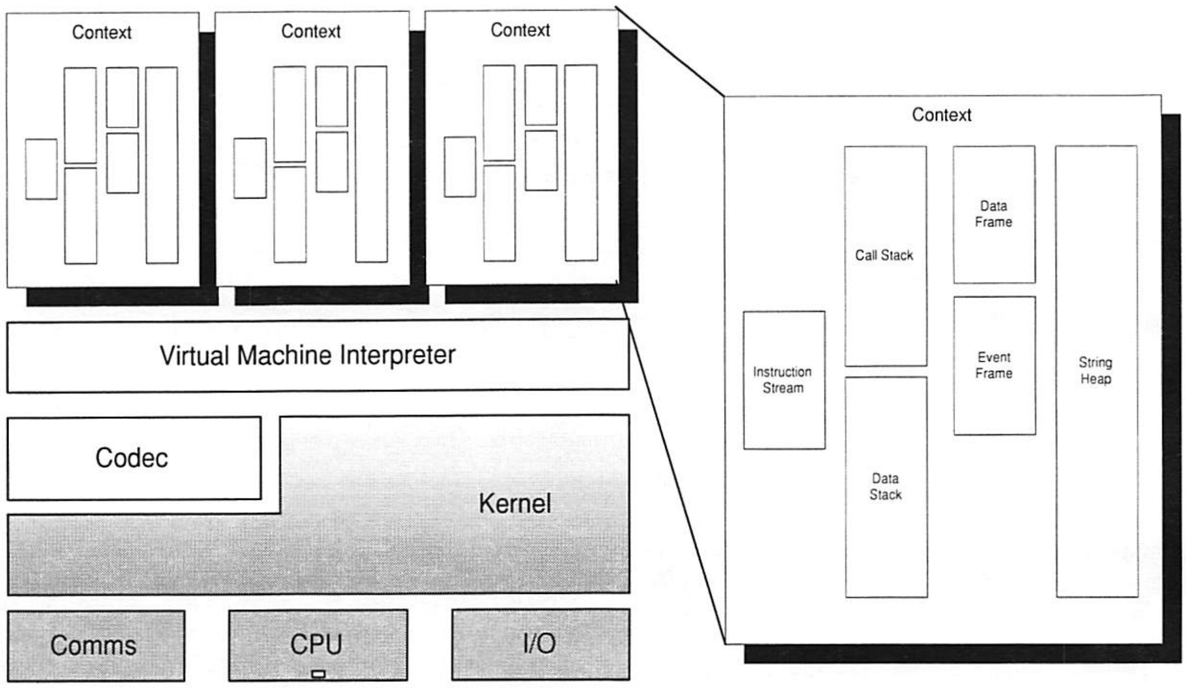
Berkeley Motes provide a consistent model for the development of smart spaces. Since Motes have a very small memory footprint, there have been several developments designed to operate in such a constrained environment.

The TinyOS[12] operating system has been developed to provide support for Motes. The nesC[10] language is a language oriented towards TinyOS applications — and TinyOS itself. TinyOS/nesC is designed to support complete static analysis of applications, including atomic operations and data race detection, to ensure reliability. A single application is linked with TinyOS and deployed as a single unit. This approach can be contrasted with the approach taken in this paper, which assumes multiple, dynamic applications and the ability to kill (and reload) misbehaving components.

The obvious advantages of using virtual machines in smart spaces has led to the development of Maté[13] for networks of Motes. There is a considerable overlap between Maté and the virtual machines described in this paper: stack-based, active messages, small footprint. However, Maté follows the general Motes philosophy: a single program and an essentially homogeneous environment allowing a single virtual machine implementation and instruction-level access to hardware.

Virtual machine generation has been used successfully with the VMGen[7] system, used to generate virtual machines for GForth. The virtual machine generator presented here shares many of the characteristics of VMGen, although VMGen performs sophisticated superinstruction generation and does not permit subsetting.

The Denali lightweight virtual machine model[23] offers a similar model to that discussed in this paper: lightweight, multiplexed virtual machines acting as monitors and sharing resources across a network. However, the focus of Denali is on providing isolated multi-



Darker blocks are more hardware-specific.

Figure 3: Generic Virtual Machine Architecture

plexing on large, general-purpose systems and the para-virtualisation techniques used in Denali would compromise the goal of a common runtime.

Also suitable for larger embedded devices is Scylla[19]. Scylla uses a register-based virtual machine that can be easily compiled into the instruction set present in larger embedded processors, such as the ARM. Scylla is oriented towards (just-in-time) compiled applications, something beyond the power of many of the resources discussed in this paper.

1.2 Overview

The paper is structured as follows: Section 2 gives a description of the generic virtual machine architecture that is supported, a stack-based virtual machine with a range of specific data stores; Section 3 describes the way a specific virtual machine is declared in an XML document; Section 4 discusses code generation from a specification to compiled virtual machine, along with some discussion of the size of the generated virtual machine and of potential optimisations; Section 5 concludes the paper.

2 Architecture

The generic virtual machine architecture is shown in Figure 3. The generated virtual machines are stack-based, for the reasons outlined in [6]: ease of code generation and lack of instruction decoding overhead. The generic architecture contains a number of elements: contexts that contain the state of a component; a virtual machine interpreter; a coder-decoder (codec) for marshalling and unmarshalling events; a platform-specific kernel and hardware support (communications, processing and I/O for sensors and actuators). Communications with the outside world, either as direct I/O or across a network link are handled in terms of events. These elements are discussed in more detail below.

2.1 Contexts

Contexts provide a complete state description of a virtual machine component. Since a resource may be managing several components, multiple contexts are supported, with the virtual machine interpreter multi-threading non-preemptively between them.

In addition to some state variables governing error handling and timing, a context consists of a number of *stores* of various types:

stack A LIFO stack. Stacks that grow upwards or downwards are supported. Two distinguished stacks are the data stack, the default stack for storing operands, and the call stack, used to manage subroutine and event management calls.

stream A stream of data or instructions. Unlike a stack, a stream is assumed to have a single direction, with each read returning the next element of the stream. The distinguished instruction steam is used to provide a stream of instructions for the interpreter.

frame An indexed data frame. Frames contain data in a fixed position that needs to be accessed by a component. A special frame is the dispatch frame, used to store pointers to subroutines that service events.

heap A garbage-collected heap for storing variable-length string or binary data.

Heaps are garbage collected by a conservative mark and sweep, non-compacting algorithm that performs stack and frame walks, looking for possible heap references[3]. Some of the easier misidentification avoidance techniques have been used[2]. Pointers are aligned and references to heap objects are given an unlikely signature, to avoid too many spurious references being identified. Ignoring compaction, while increasing the risk of fragmentation, removes the need for a separate object table.

Stores have an associated data type (eg. 32-bit integer, program instruction pointer) that can be used to translate data moving between the various stores. Stores can be designated as read-only, providing a hint that the store could be placed in flash memory.

2.2 The Loader

Contexts have a loader format that allows a context to be transmitted over the network as a stream of binary data. The loader format lists the various stores that need to be loaded, including the instruction stream.

The loader format is designed to minimise network message size and initialisation code. No relocation is needed, since all addresses are relative to the start of the store. Each store can be installed with the bottom (or top) part of the store pre-initialised; even a stack or heap can be pre-initialised before the program starts. Parts of the store that are not pre-initialised are initialised to a default value, to ensure application isolation. Each store is supplied with an expected size, the size of any pre-initialised data and some information on the expected type and data-type of the store, for basic consistency checking.

Installing a context involves allocating space for the various stores and initialising them from whatever data is supplied to the context. The context can then be added to the scheduling list for the virtual machine interpreter.

2.3 The Interpreter

The virtual machine interpreter is responsible for managing the scheduling of contexts and kernel functions, and the execution of a context's instruction stream.

The interpreter executes in a non-preemptive fashion, with certain instructions causing the interpreter to yield. A specification option also allows yielding after a fixed number of instructions, ensuring good behaviour in an untrusted environment. External events, such as timers, sensor triggers or network messages are handled by buffering the incoming data until the interpreter is prepared to process it as an event while switching contexts.

Atomic sections of code can be created by preventing yielding. If yielding is not forced after a fixed number of instructions, then atomic sections simply consist of sequences of non-yielding instructions. In this case the code generator for the virtual machine programs needs to be trusted to perform a suitable yielding analysis. If yielding is forced after a fixed number of instructions, it is possible for a yield to occur within an atomic section. A specification option allows a flag to be included in the virtual machine that will cause the program to be immediately rescheduled after yielding. If the flag is not released after a specified time period, then the program is assumed to be malfunctioning and is terminated.

This approach can be contrasted to that of nesC[10], where explicit atomic actions and data race detection is built into the language. The approach taken here pushes the complications of managing hardware-specific functions onto the kernel developer (see Section 2.5) and the issues of ensuring yielding onto the code generator.

2.4 Events and the Codec

Communication with the outside world, either via network messages or through the resource's I/O facilities is handled via events. The virtual machine supports a set of named events, each with an explicit set of parameters.

An outgoing event is sent either across the network or to a service routine in the kernel, where it is applied to the resource. An incoming event is handled by a service routine in a virtual program. The service routine is supplied the event arguments and is expected to capture the arguments and handle any specific responses.

The codec (coder-decoder) is responsible for translating events from/to the stores of a context. To code an outgoing event, the event parameters are retrieved from the data stack and then either marshalled into a message or passed on to the kernel. To decode an incoming event, the incoming message or hardware event is unmarshalled and the event parameters pushed onto the data stack. A call to a service routine, chosen from the dispatch frame, is then inserted into the context and the context is scheduled. When the context is next processed, it will interpret the service routine before returning to the main program thread. To prevent reentrant events, event handling routines need to be atomic (see Section 2.3).

The approach taken is similar to that of active messages[22]. Each message is identified by a type and decoded according to the supplied type. Decoding is done by the codec routines, rather than by the service routines. By decoding the message early, the message buffer can by recycled immediately, rather than needing

```
<type ID="int" prefix="i" stack="data-stack" cell="int32"/>

<stack ID="dataStack" name="sp" type="int" default="true" defaultSize="16"/>

<instruction ID="add">
  <description>Add the two top entries on the stack.</description>
  <argument>v1</argument>
  <argument>v2</argument>
  <result>v</result>
  <operation target="java">
    v = v1 + v2;
  </operation>
  <operation target="c">
    v = v1 + v2;
  </operation>
</instruction>

<event ID="reading">
  <description>Notification of a reading from a sensor.</description>
  <argument>sensor</argument>
  <argument>data</argument>
</event>
```

Figure 4: Example Virtual Machine Declarations

to wait for each context to decode the message individually.

Message addressing is via UUIDs[11]. Each context is given a UUID, allowing simple point-to-point messaging, as well as broadcast.

From the point of view of a context, events that cause messages are indistinguishable from direct hardware events. Treating the two uniformly makes translation between a resource with direct access to sensors and other elements and a resource that needs to make use of other resources relatively straightforward.

2.5 The Kernel

The kernel is the interface between the hardware of a resource and the virtual machine. The kernel is responsible for:

- memory management;
- communications and connection management;
- interfaces to directly implemented events;
- direct output to hardware;
- managing input (synchronous and asynchronous) from hardware; and
- low-level timing

The kernel and virtual machine interpreter run under a single thread. Other threads — or interrupt routines — may handle aspects of I/O and communications. These threads are invisible to the interpreter. The kernel is polled by the virtual machine interpreter for any events while switching contexts. If there are no active contexts, the kernel is responsible for waiting for an event or timeout for the interpreter to process.

3 Virtual Machine Specification

A virtual machine is specified in an XML document. The use of XML allows both ease of use and the wide range of XML tools and technologies to be applied to the specification. The specification allows a stack-based virtual machine to be generated. The essential elements of a specification, shown in Figure 4, are:

type declarations Type declarations allow the creation of logical types, such as `int`. Logical types can be associated with particular primitive types, such as `int32` for 32-bit integers and default store locations.

store declarations Store declarations describe the stacks, heaps and other elements that the virtual machine manipulates.

instructions Instruction definitions describe the input and output arguments of the instruction, along with the stores that the arguments come from and go to. Repeated argument names are assumed to refer to the same value. Code implementing the instruction in the target language (Java or C) can also be given.

events Event definitions are similar to instruction definitions, except that no implementing code is supplied. The implementation of the event is either as a direct kernel function or as a message sent to another resource.

```
<subset>
  <description>Exclude strings</description>
  <include type="store">.*</include>
  <include type="instruction">.*</include>
  <include type="event">.*</include>
  <exclude type="store">strings</exclude>
  <exclude type="instruction">dups</exclude>
  <exclude type="instruction">appends</exclude>
  <exclude type="instruction">str</exclude>
  <exclude type="event">message</exclude>
</subset>
```

Figure 5: Example Subset Declaration

In addition to the basic virtual machine definition, a separate XML document contains a subset declaration for the virtual machine. An example subset declaration is shown in Figure 5. The subset declaration lists those instructions, events and stores that are to be implemented. The subset declaration also, in the case of events, defines them to be direct or message events. Subset elements can be defined either by inclusion or exclusion. For conciseness, the inclusions and exclusions use regular expressions to match store, instruction and event names.

4 Virtual Machine Generation

The virtual machine generation process is shown in Figure 6. A virtual machine specification and subset declaration are fed into the generator. The generator then analyses the virtual machine and generates a series of source code files for Java and C that implement the subset virtual machine. The source files are then compiled and linked against a standard library of support functions and classes. An assembler is also generated. Sample declared instructions and generated C code is shown in Figures 7 and 8.

The complete virtual machine is analysed and instruction codes, event codes and stores are allocated before subsetting. By analysing the complete virtual machine, a subset virtual machine is guaranteed to be compatible with any superset implementation.

Code generation makes extensive use of the Visitor pattern[9]. Each virtual machine construct (type, instruction, store, event, etc.) is represented by an object. A language-specific generator is then used to generate appropriate code.

Superinstruction analysis and generation[7] is not performed. The trade-off in a memory-constrained environment between virtual machine size, on one hand, and code size and speed, on the other hand, is difficult to manage. The aim of the generator is to generate multiple virtual machines, all providing a subset of a common runtime.

4.1 Java Code Generation

Java code generation is relatively straightforward. A separate class file for each element of the virtual machine shown in Figure 3 is generated, along with interfaces for common elements, such as instruction codes. Abstract superclasses provide any common functionality that is needed.

The interpreter uses a large switch statement to decode instructions. For each instruction, arguments are gathered from the various stores and placed in temporary variables. Any implementation code that is part of the declaration is then executed. Any results are then returned to the appropriate stores.

The generated virtual machine interpreter moves commonly used context elements (stack pointers, store arrays) to temporary variables while the context is being executed. These variables are replaced whenever the interpreter cycle for that context finishes or when an instruction with side-effects — such as an event send — is executed. The more sophisticated stack caching techniques, discussed in [6], are not implemented, although implementing them would clearly improve performance and caching behaviour.

The UUID method of addressing has proved cumbersome. It is difficult to handle 128-bit objects efficiently without generating large amounts of code, special instructions and special stores. A local context identifier that fits the natural data size of the virtual machine would seem to be more useful, at the expense of more management complexity at higher levels.

4.2 C Code Generation

The C code generator generates code that is very similar to the generated Java code. The main difference between the two generators is that structs, rather than classes, are used for data structures, with functions taking the structs as arguments. Library code is in the form of individual functions, rather than abstract classes. C, rather than C++, is generated, so that a minimalist approach can be taken to object construction and destruction.

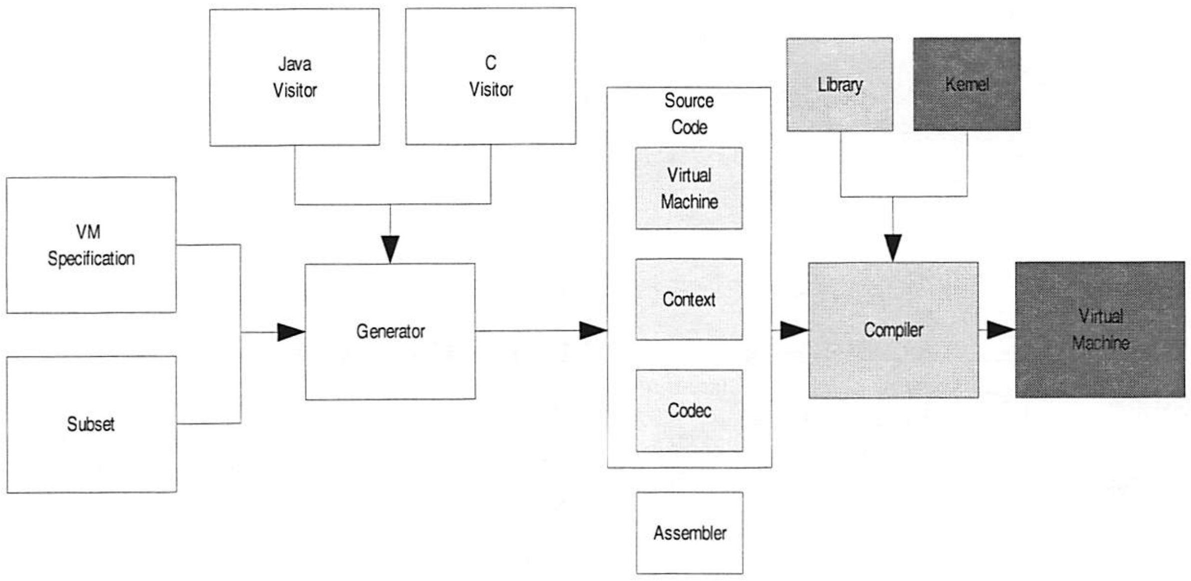


Figure 6: Virtual Machine Generation

```
<instruction ID="dup">
  <argument>v</argument>
  <result>v</result>
  <result>v</result>
</instruction>
```

(a) instruction declaration

```
case I_DUP:
  if (sp + 1 > dataStackSize || sp < 1)
    goto stack_error;
   _temp1 = dataStackData[sp++];
   dataStackData[--sp] = _temp1;
   dataStackData[--sp] = _temp1;
   break;
```

(b) generated code

Figure 7: Code generation for the dup instruction

```
<instruction ID="str">
  <argument>v</argument>
  <result heap="strings">r</result>
  <operation target="java">
    r = Integer.toString(v);
  </operation>
  <operation target="c">
    itoa(scratch_buffer, SCRATCH_BUFFER_SIZE, v);
    r = scratch_buffer;
  </operation>
</instruction>
```

(a) instruction declaration

```
case I_STR:
  if (sp + 1 > dataStackSize)
    goto stack_error;
  _temp1 = dataStackData[sp++];
  itoa(scratch_buffer, SCRATCH_BUFFER_SIZE, _temp1);
  _temp6 = scratch_buffer;
  _temp2 = _temp6 == NULL ? 0 :
    heap_storeString(context->strings, _temp6);
  if (_temp2 < 0)
    goto heap_error;
  dataStackData[--sp] = _temp2;
  break;
```

(b) generated code

Figure 8: Code generation for the `str` instruction

	Library Code				
Class	Java		C		Description
	Full	No Strings	Full	No Strings	
Basic Heap	0	0	643	643	Core heap management
VM Base	1259	1250	690	690	Common virtual machine functionality
Kernel Base	29	29	80	80	Basic kernel functionality
Codec Base	268	268	916	740	Common coder-decoder functionality
Connection	1335	1158	531	531	Communications management, marshalling and unmarshalling
Heap Manager	1705	0	878	0	Garbage-collected heap management
Loader	1365	1078	955	741	Context unmarshalling and loading
UUID	539	539	68	68	UUID implementation
	6500	4331	4761	3493	

	Generated Code				
Class	Java		C		Description
	Full	No Strings	Full	No Strings	
Codec	1252	1134	1215	839	Generated coder-decoder
Context	786	530	787	525	Generated context
Kernel	660	660	801	801	Kernel for 3 LEDs, a temperature sensor and a heat pump
VirtualMachine	1777	1527	1913	1430	Generated virtual machine
	4475	3851	4716	3595	

Table 1: Code Sizes for a Generated Virtual Machine

The C virtual machine interpreter needs to do a great deal more bounds checking than the Java interpreter. Stacks, for example, may not overrun their boundaries — something guaranteed by the Java virtual machine.

4.3 Code Size

The code generated is relatively compact. Table 1 shows the relative code sizes for a simple virtual machine with and without string handling. The Java code was generated by the Sun 1.4.2_01 javac compiler. The C code was generated for a Pentium 4 processor by gcc 3.3.2 with the -Os option. Total size is 9–11k bytes of code for the virtual machine with string handling and 7–8k for the same machine without string handling.

The full virtual machine contains 29 instructions, 6 events, a data stack, a call stack, a data frame, a dispatch frame, a string heap and an instruction stream. The stringless virtual machine contains 25 instructions, 6 events, a data stack, a call stack, a data frame, a dispatch frame and an instruction stream. The underlying resource is a simple resource with 3 3-colour LEDs, a temperature sensor and a heat pump.

String handling increases the size of the generated virtual machine considerably. Clearly, a heap manager is needed, which increases code size. However, string management tends to be more complex in general, requiring specialised marshalling and unmarshalling and more complex instruction implementations. The method size in both the Connection and Codec classes increases by approximately 50% whenever string handling is needed. More importantly, given the small amount of RAM available, string handling requires the allocation of blocks of memory to act as a heap.

The network management and message passing parts of the virtual machine take up a significant part of the total memory footprint. Message and program transmission can be considered a relatively rare event — or, at least, it should be, if energy consumption is to be taken into account — in which case its influence on caching and power consumption (see Section 5) can be regarded as negligible. However, it would be a good thing, on principle, to reduce the amount of code needed for such an operation. At present, marshalling is handled by dedicated routines, one to each type of message. An alternative is to try an data-driven, interpreter-based approach[5]. If there a large number of events, this approach looks attractive.

An assembled program takes up little space. Table 2 summarises the context sizes, in the network deliverable loader format (see Section 2.2), for a number of simple programs.

The sizes shown in Table 2 show the minimum amount of information needed to initialise a context. Installed contexts usually take up more space within the resource: stacks need enough room to grow and heaps usually need additional space for new blocks of data.

Program	Size (bytes)	Description
ChangeReport	134	Polling report of sensor change
EventReport	116	Event-driven report of sensor change
Chaser	166	LED chaser
AirCon	167	Simple airconditioning

Table 2: Assembled Application Code Sizes

5 Conclusions and Further Work

The diversity and complexity of heterogeneous smart spaces, coupled to the stringent restrictions on resource usage that networks of small embedded devices imply, presents a considerable software engineering challenge. The sort of component reuse strategies that have become common in commercial programming environments will also need to be applied to smart spaces, if smart spaces are to become general-purpose, commercial environments. The use of virtual machines provides a method for distributing generic functionality across a wide range of resources.

There are a number of virtual machine optimisations and improvements that could be undertaken. These optimisations are discussed in Sections 4.1 and 4.3. In particular, code-size optimisations can be expected to play an important part in reducing the size of the generated virtual machine. An advantage to using a generator is that any optimisations that are made will propagate to any newly generated virtual machine, rather than requiring hand-optimisation.

Energy consumption and power management is a major concern in the space of small embedded devices, with memory access a significant source of energy consumption. Testing of the energy consumption of Java virtual machines in the Itsy pocket computer suggests that there is the order of a 50% penalty in energy consumption when interpretation is used, instead of a just-in-time compiler[8]. There is an order of magnitude difference between cache memory access and external memory access, however[1]. If the virtual machine interpreter — or a subset of frequently used instruction implementations — and a context could be fitted into cache memory, the energy costs could be significantly reduced. The compression effect of virtual machine instructions would then serve a useful purpose in allowing a component to be entirely cached.

Generating virtual machine subsets allows a common runtime environment to be imposed on the diverse array of resources that make up a heterogeneous smart space. Using a generator allows virtual machines to be quickly generated for new resources and to try new instruction sets. The generated virtual machine is relatively compact, although there is considerable room for improvement.

References

[1] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions on Embedded Computing Systems*, 2(1):5–32, February 2003.

[2] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 197–206, Albuquerque, New Mexico, June 1993.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, September 1988.

[4] Crossbow: Wireless sensor networks, 2003. `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[5] K.V. Dyshlevoi, V.E. Kamensky, and L.B. Solovskaya. Marshalling in distributed systems: Two approaches, June 1997. `http://www.ispras.ru/~microrb/papers/index.html`.

[6] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, La Jolla, California, June 1995.

[7] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software – Practice and Experience*, 32(3):265–294, 2002.

[8] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 252–263, Santa Clara, California, June 2000.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, California, June 2003.

[11] The Open Group. *DCE 1.1: Remote Procedure Call*, October 1997. Standard C706, `http://www.opengroup.org/onlinepubs/009629399/toc.pdf`.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, Massachusetts, November 2000.

[13] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85–95, San Jose, California, October 2002.

[14] Howard Lovatt, Geoff Poulton, Don Price, Mikhail Prokopenko, Philip Valencia, and Peter Wang. Self-organising impact boundaries in ageless aerospace vehicles. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, pages 249–256, June 2003.

[15] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, 1993.

[16] Palm, inc., 2003. `http://www.palm.com`.

[17] Doug Palmer. Declarative application programming in smart spaces. Technical Report 03/88, CSIRO Mathematical and Information Sciences, January 2003.

[18] Smartlands, 2003. `http://www.smartspaces.csiro.au/applic/smart-lands.htm`.

[19] Phillip Stanley-Marbell and Liviu Iftode. Scylla: A smart virtual machine for mobile embedded systems. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00)*, pages 41–50, Monterey, California, December 2000.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[21] Ken Taylor and Doug Palmer. Applying enterprise architectures and technology to the embedded devices domain. In *Proceedings of the Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing (WICAPUC)*, number 21 in Conferences in Research and Practice in Information Technology, Adelaide, Australia, February 2003.

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. Technical Report USB/CSD 92/#675, University of California, Berkeley, March 1992.

[23] Andrew Whitaker, Marianne Shaw, and John D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2002.

MCI-Java: A Modified Java Virtual Machine Approach to Multiple Code Inheritance

Maria Cutumisu, Calvin Chan, Paul Lu and Duane Szafron
Department of Computing Science, University of Alberta
{meric, calvinc, paullu, duane}@cs.ualberta.ca

Abstract

Java has multiple inheritance of interfaces, but only single inheritance of code via classes. This situation results in duplicated code in Java library classes and application code. We describe a generalization to the Java language syntax and the Java Virtual Machine (JVM) to support multiple inheritance of code, called MCI-Java. Our approach places multiply-inherited code in a new language construct called an *implementation*, which lies between an interface and a class in the inheritance hierarchy. MCI-Java does not support multiply-inherited data, which can cause modeling and performance problems. The MCI-Java extension is implemented by making minimal changes to the Java syntax, small changes to a compiler (IBM Jikes), and modest localized changes to a JVM (SUN JDK 1.2.2). The JVM changes result in no measurable performance overhead in real applications.

1 Introduction

Three distinct language concepts are used in object-oriented programs: interface, code and data. The motivation for separate language mechanisms to support these concepts has been described previously [14]. The goal of the research described in this paper is to explicitly support each of these three mechanisms in an extended Java language, to evaluate the utility of concept separation, and to potentially increase demand for separate language constructs in future languages.

Researchers and practitioners commonly use the terms *type* and *class* to refer to six different notions:

a real-world concept (**concept**)
a programmatic interface (**interface**)
the code for an interface (**implementation**)
an internal machine data layout (**representation**)
a factory for creation of instances (**factory**)
a maintainer of the set of all instances (**extent**)

Unfortunately, most object-oriented programming languages do not separate these notions. Each language models the **concept** notion by providing language constructs for various combinations of the other five notions. Java uses *interface* for **interface**. However, it combines the notions of **implementation**, **representation** and **factory** into a *class* construct. Smalltalk and C++ have less separation. They use the same *class* construct for **interface**, **implementation**, **representation** and **factory**.

In this paper we will focus on general-purpose programming languages, so we will not discuss the **extent** notion, which is most often used in database programming languages [15]. We also combine **concept** with **interface**, to enforce encapsulation. Finally, we combine **representation** and **factory**, since **representation** layout is required for creation. This reduces the programming language design space to three dimensions:

An *interface* defines the legal operations for a group of objects (**interface**) that model a **concept**.
An *implementation* associates generic code with the operations of an interface (**implementation**) without constraining the data layout.
A *representation* defines the data layout (**representation**) of objects for an **implementation**, provides data accessor methods and a mechanism for object creation (**factory**).

We use the generic term *type* to refer to an *interface*, an *implementation* or a *representation*.

Inheritance allows properties (*interface*, *implementation* or *representation*) of a type to be used in child types called its *direct subtypes*. By transitivity, these properties are inherited by all *subtypes*. If type `B` is a subtype of type `A`, then `A` is called a *supertype* of type `B`. If a language restricts the number of direct supertypes of any type to be one or less, the language has *single inheritance*. If a type can have more than one direct supertype, the language supports *multiple inheritance*.

Interface inheritance allows a subtype to inherit the *interface* (operations) of its supertypes. The principle of *substitutability* states that if a language expression contains a reference to an object whose static type is `A`, then an object whose type is `A` or any subtype can be used. Interface inheritance relies only on substitutability and does not imply code or data inheritance.

Table 1 Support for single/multiple inheritance and concept separation constructs in some existing languages.

Language	*interface*	*implementation*	*representation*
Java	multiple / interface	single / class	single / class
C++	multiple / class	multiple / class	multiple / class
Smalltalk	single / class	single / class	single / class
Eiffel	multiple / class	multiple / class	multiple / class
Cecil/BeCecil	multiple / type	multiple / object	multiple / object
Emerald	implicit multiple / abstract type	none / object constructor	none / object constructor
Sather	multiple / abstract class	multiple /class	multiple / class
Lagoona	multiple / category	none / methods	single / type
Theta	multiple / type	single / class	single / class

Implementation (code) inheritance allows a type to reuse the *implementation* (binding between an operation and code) from its parent types. Code inheritance is independent of data representation, since many operations can be implemented by calling more basic operations (e.g. accessors), without specifying a representation, until the subtypes are defined. Java and Smalltalk only have single code inheritance, but C++ has multiple code inheritance.

Representation (data) inheritance allows a type to reuse the *representation* (data layout) of its parent types. This inheritance is the least useful and causes considerable confusion if multiple data inheritance is allowed. Neither Java nor Smalltalk support multiple data inheritance, but C++ does.

Table 1 shows the separation and inheritance characteristics of several languages: Java, C++, Smalltalk, Eiffel [17], Cecil [6] and its descendant BeCecil [5], Emerald [20], Sather [22], Lagoona [12] and Theta [9]. This list is not intended to be complete. A more general and extensive review of type systems for object-oriented languages has been compiled [15]. Although there is growing support for the separation of *interface* from *implementation/representation*, the concepts of *implementation* and *representation* are rarely separated at present. We intend to change this. The major research contributions of this paper are:

- The introduction of a new language construct called an *implementation* into the Java programming language. This construct completely separates the two orthogonal concepts of *implementation* (code) and *representation* (data).
- A new multi-super call mechanism that generalizes current Java semantics, rather than using C++ multi-super semantics.
- The first implementation of multiple code inheritance in Java, based on localized modifications to the SUN JDK 1.2.2 JVM, along with minor changes to the syntax of Java and to the IBM Jikes 1.15 compiler. Existing programs still work and suffer no performance penalties.
- A demonstration that multiple code inheritance reduces duplicated and similar code, so program construction and maintenance are simplified.

Our modified multiple code inheritance compiler (*mcijavac*), modified JVM (*mcijava*), the code for the scenarios in this paper, and the code for the `java.io` example are available on-line as MCI-Java [16].

2 Motivation for Multiple Code Inheritance in Java

In this Section we motivate the use of multiple code inheritance using some classes from the `java.io` library. Java currently supports multiple interface inheritance. Consider the Java class `RandomAccessFile` (from `java.io`) that implements the interfaces `DataInput` and `DataOutput`, as shown in Figure 1[1]. Since Java supports substitutability, any reference to a `DataInput` or `DataOutput` can be bound to an instance of `RandomAccessFile`.

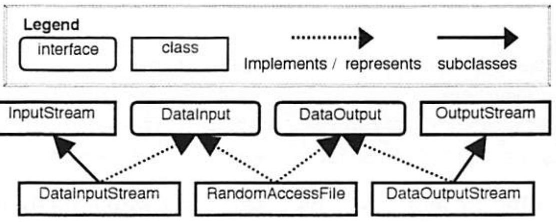


Figure 1. The inheritance structure of some classes and interfaces from the `java.io` library.

However, Java does not support multiple code inheritance. Much of the code that is in `RandomAccessFile` is identical or similar to code in

[1] There are actually two other classes in `java.io`, `FilterInputStream` and `FilterOutputStream`, that are not included in Figure 1, Figure 2, or Figure 5. They have been omitted for simplicity and clarity, since they do not affect the abstractions described in this paper.

DataInputStream and DataOutputStream. Although it is possible to refactor this hierarchy to make RandomAccessFile a subclass of either DataInputStream or DataOutputStream, it is not possible to make it a subclass of both, since Java does not support multiple code inheritance.

This causes implementation and maintenance problems. One common example is that duplicate code appears in several classes. This makes programs larger and harder to understand. In addition, code can be copied incorrectly or changes may not be propagated to all copies. There are many examples of code copying errors in various contexts. For example, code is often cloned (cut-and-pasted) in device drivers for operating systems [7]. If a bug is found in the repeated code, a fix must be applied to each clone. However, if the same code is refactored into a single *implementation* in an object-oriented inheritance hierarchy, then any bug fix or new functionality would only have to be done once.

Two alternative techniques for reducing code duplication are *mixins* [1] and *traits* [21]. These approaches are discussed in Section 9, where they are contrasted with our multiple code inheritance technique.

2.1 Duplicate Method Promotion

The methods writeFloat(float) and writeDouble(double) are examples of duplicate methods that appear in both DataOutputStream and RandomAccessFile. There are also four methods that have identical code in DataInputStream and RandomAccessFile.

Once these duplicate methods have been found, how can code inheritance be used to share them? Figure 1 shows that we need a common ancestor type of DataInputStream and RandomAccessFile to store the four common read methods and a common ancestor type of DataOutputStream and RandomAccessFile to store the two common write methods. To share code, this ancestor cannot be an interface. It also cannot be a class, since we would need multiple code inheritance of classes and Java does not support it. In fact, it should be a multiple inheritance *implementation*. Figure 2 shows the common code factored into two *implementations*: InputCode and OutputCode. In this approach, an *implementation* provides common code for multiple classes.

The benefit of using *implementations* to promote *duplicate methods* may seem questionable to re-use only six methods. However, it is not only such duplicate methods that can be promoted higher in the inheritance hierarchy. Multiple code inheritance can also be used to factor some non-duplicate methods, if they are abstracted slightly. We have used three additional code promotion techniques to factor non-duplicate methods.

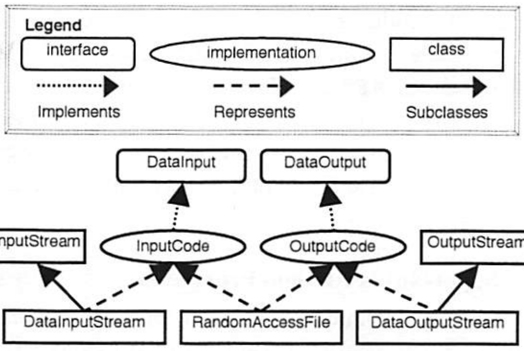


Figure 2. Adding *implementations* to Java, for multiple code inheritance.

2.2 Prefix Method Promotion

The first technique is called *prefix method promotion*. It applies when a class dependent computation is done at the start of the method and the rest of the method is identical. Consider the readByte() methods shown in Figure 3, from classes DataInputStream and RandomAccessFile. These methods differ only by a single line of code.

```
// This method is in DataInputStream
public final byte readByte() throws IOException
{
   int ch = this.in.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}

// This method is in RandomAccessFile
public final byte readByte() throws IOException
{
   int ch = this.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}

// This method replaces the previous two, and
// is in InputCode
public final byte readByte() throws IOException
{
   Source in = this.source();
   int ch = in.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}
```

Figure 3. An example of using the prefix technique to promote methods.

Figure 3 shows a single common method promoted to an *implementation* called InputCode, which replaces both. This abstraction requires the creation of a new interface, Source, that contains one abstract method, read(). It also requires a method called source() to be declared in InputCode. A default method with code, `return this`, is created in `InputCode` and it is inherited by class `RandomAccessFile`. The implementation of this method is overridden in class `DataInputStream` with the code, `return this.in`. This prefix technique can be used to promote seven other methods in the same two classes.

2.3 Super-Suffix Method Promotion

The second technique is called *super-suffix method promotion*. It can be used to move similar methods from `DataOutputStream` and `RandomAccessFile` to the common *implementation*, `OutputCode`. Consider methods for `writeChar(int)` that appear in classes `DataOutputStream` and `RandomAccessFile`, as shown in Figure 4.

```
// The original method in DataOutputStream
public final void writeChar(int v) throws
IOException {
   this.out.write((v >>> 8) & 0xFF);
   this.out.write((v >>> 0) & 0xFF);
   incCount(2);
}

// The original method in RandomAccessFile
public final void writeChar(int v) throws
IOException {
   this.write((v >>> 8) & 0xFF);
   this.write((v >>> 0) & 0xFF);
}

// The method in OutputCode that replaces the
// one in RandomAccessFile and does most of the
// computation for the one in DataOutputStream
public final void writeChar(int v) throws
IOException {
   Sink out = this.sink();
   out.write((v >>> 8) & 0xFF);
   out.write((v >>> 0) & 0xFF);
}

// The new method in DataOutputStream
public final void writeChar(int v) throws
IOException {
   super(OutputCode).writeChar(v);
   incCount(2);
}
```

Figure 4. Examples of super-suffix methods that can be reused in different classes.

To replace these methods by a common method, we substitute each first line by a common abstracted line, analogous to the previous example. This abstraction requires the creation of a new interface, `Sink`, that contains one abstract method, `write(int)`. It also requires a method called `sink()` to be declared in `OutputCode`.

However, there is another problem that must be solved before we can promote `writeChar(int)`. This method has an extra line of code in class `DataOutputStream`, which does not appear in class `RandomAccessFile`. Fortunately, we can promote all lines except for this *suffix* line into a common method in `OutputCode`. This eliminates `writeChar(int)` from `RandomAccessFile`. However, in `DataOuputStream` we need to include the missing last line using a classic refinement technique that makes the super call shown in Figure 4.

Note that `super(OutputCode)` is **not** standard Java. It calls a method in the *superimplementation*, `OutputCode`, instead of calling a method in a superclass. In general, since there may be multiple immediate *superimplementations*, the super call must be qualified by one of them. This is one of the standard approaches to solving the super ambiguity problem of multiple inheritance and it will be discussed later in this paper. We can use this super-suffix technique to promote a total of six similar methods that appear in `DataOuputStream` and `RandomAccessFile`.

2.4 Static Method Promotion

The third technique for promoting non-duplicate methods is called *static method promotion*. For example, both `DataInputStream` and `RandomAccessFile` implement `readUTF()`. The class implementers must have realized that the two implementations were identical, so rather than repeating the code, they created a static method called `readUTF(DataInput)` and moved the common code to this static method in class `DataInputStream`. Then they provided short one line implementations of `readUTF()` in `DataInputStream` and `RandomAccessFile` that call the static method. Now that we have provided a common code repository (`InputCode`) that both `DataInputStream` and `RandomAccessFile` inherit from, we can eliminate the static method by moving its code to `InputCode` and eliminate the short methods that call this common code, since both classes now share this common instance method. This is an example where we did not actually remove repeated code. Instead, we replaced one code sharing abstraction (static sharing), that can cause maintenance problems, by a better code sharing mechanism (inheritance).

We conducted an experiment to determine how much code from the stream classes of the `java.io` libraries could be promoted, if Java supported multiple code inheritance. Table 2 and Table 3 show a summary of the method promotion and lines of code promotion respectively for each of our code promotion techniques.

Table 2 Method decrease in the Java stream classes using multiple code inheritance. The number marked with a * indicates that all lines of code (except for one line) in the method were promoted, so a single line method remained.

Class	duplicate	prefix	super-suffix	static elimination	total promoted	method decrease
DataInputStream	4 of 19	8 of 19	0 of 19	1+1*	14 of 19	74%
DataOutputStream	2 of 17	0 of 17	6* of 17	0	8 of 17	47%
RandomAccessFile	6 of 45	8 of 45	6 of 45	1	21 of 45	47%

Table 3 Executable code line decrease in the Java stream classes using multiple code inheritance. All extra lines of executable code from the extra classes, Source and Sink, are also included in the third column.

Class	initial lines	extra lines for prefix, super-suffix and static elimination	net lines after all code promotion techniques	line decrease
DataInputStream	127	2	42	67%
DataOutputStream	84	7	67	20%
RandomAccessFile	158	0	93	41%

For example, from Table 2 we see that there are 19 methods in class `DataInputStream`. Of these 19 methods, 4 were promoted since there are duplicate methods in class `RandomAccessFile`. An additional 8 methods out of 19 were promoted using the prefix technique illustrated in Figure 3.

Finally, one method was eliminated using static elimination. The instance method was promoted to `InputCode` and one static method was reduced from 40 lines to 1 line. Table 2 shows that the super-suffix technique resulted in 6 promoted methods out of 45 in class `RandomAccessFile`. The corresponding 6 methods in `DataOutputStream` (marked by an asterisk in Table 2) were not completely promoted. A shorter method was retained to make the suffix super call and to execute one or more additional lines of code.

For the line counts, we only counted executable lines and declarations, not comments or method signatures. However, more important than the size of the reductions is the lower cost of understanding and maintaining the abstracted code. Note that even though most of the method bodies of six methods move up from `DataOutputStream` to `OutputCode`, small methods remain that make super calls to these promoted common "prefix" methods. In Table 3, the third column indicates the lines that were added for an abstraction (`Sink out = this.sink();`) or a multi-super call (`super(OutputCode).writeChar(v);`). All executable lines of code in *implementations* `InputCode` and `OutputCode` are included in column 4 of Table 3.

Note that this abstraction required the creation of another new interface, `Source`, which is analogous to the interface, `Sink`, which was described earlier. The resulting inheritance hierarchy for the Stream classes is shown in Figure 5.

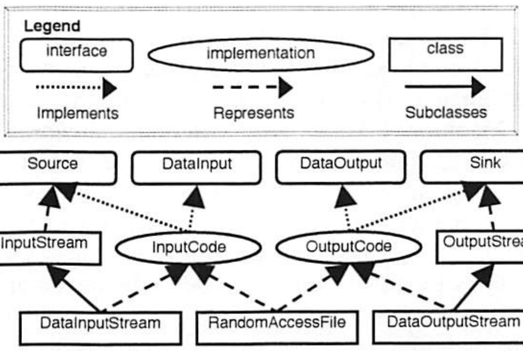


Figure 5. The revised Stream hierarchy to support multiple code inheritance.

The new interfaces `Source` and `Sink` only contain declarations of the `read()` and `write()` methods, so they contain no lines of executable code. They only exist so that they can be used as the static type of the variables `in` and `out` in the *implementations* `InputCode` and `OutputCode`.

Table 2 and Table 3 show that the use of multiple inheritance in Java can result in a significant reduction in the number of duplicate lines of code in library classes. This reduction can result in fewer errors during library maintenance and library extension and can therefore reduce maintenance costs [4].

3 Supporting *Implementations* in Java

Since Java has no concept of an *implementation*, we have three choices as to how to introduce it into Java: as a class (probably abstract), as an interface, or as a new language feature. We actually need to make this decision twice: once at the source code level and once at the JVM level. It is not necessary for the choices at these two levels to be the same.

At the source code level, an abstract class seems to be an obvious choice to represent an *implementation*. However, Section 2 clearly indicates the utility of multiple code inheritance. If *implementations* were represented by classes (abstract or concrete), we would need to modify Java to support multiple inheritance of classes. This would have the undesirable side-effect of providing multiple data inheritance, since classes (even abstract classes) are also used for data. Interfaces have the multiple inheritance we want but, if we use interfaces to represent *implementations* at the source code level, we would lose the use of interfaces for their original intent – specifications with no code.

Our solution is to introduce a new language construct, called an *implementation* at the source code level. However, at the JVM level, we decided to make use of the fact that interfaces already support multiple inheritance. Therefore, we did not introduce a new language concept at the JVM level. Instead, we generalized interfaces to allow them to contain code.

To implement our solution, we made independent localized changes to the compiler and to the Java Virtual Machine (JVM). Our compiler (*mcijavac*) compiles each *implementation* to an interface that contains code in the .class file. Our modified JVM (*mcijava*) supports execution of code in interfaces and multiple code inheritance. In addition, the JVM modifications to support multiple code inheritance are executed at load-time and the changes that affect multi-super are call-site resolution changes. Therefore, the performance of our modified JVM is indistinguishable from the original JVM. In fact, the SUN JDK 1.2.2 JVM uses an assembly-language module for fast dispatch and no changes were made to this module so the fast dispatch was preserved.

Our approach decouples language syntax changes from the JVM support required for code in interfaces and multiple code inheritance. For example, someone could propose a different language construct and syntax at the source code level and make different compiler modifications. In fact, our first implementation used a source-to-source translation approach with standard Java syntax and special comments to annotate interfaces that should be treated as *implementations* [8].

As long as a compiler or translator produces code in interfaces, our modified JVM can be used to execute the code. Similarly, someone can provide an alternate JVM that supports code in interfaces and use our language syntax and compiler to support *implementations*.

Although *implementations* support multiple code inheritance, they do not support multiple data inheritance, since they cannot contain data declarations. Multiple data inheritance causes many complications in C++. For example, if multiple inheritance is used in C++, an offset for the `this` pointer must be computed at dispatch time [11]. This is not necessary for multiple code inheritance. At first glance, it may appear that the opportunities for multiple code inheritance without multiple data inheritance are few. However, examples such as the one in Section 2 exist in the standard Java libraries and many application programs.

4 The Semantics of *Implementations*

To support *implementations*, we made two fundamental changes to the language semantics: the first to support multiple code inheritance and the second to support multi-super calls.

4.1 Semantics of Multiple Code Inheritance

The twenty-two scenarios and sub-scenarios in Figure 6 represent the common situations for inheriting code from *implementations*, including multiple code inheritance (note the Legend at the top of the Figure). The circled numbers and the letter ϕ can be ignored for now, since they are related to JVM modifications that are discussed in Section 8. Other more complex scenarios can be composed from these scenarios. When the method `alpha()` is shown in a class or *implementation*, the scenario also holds if that method is inherited from a parent type. For example, in scenario 5, the `alpha()` method in class `A` may actually be inherited from a parent class or *implementation*. The semantics are consistent, regardless of whether a method is declared explicitly in a type or inherited from a supertype.

Some scenarios have two sub-scenarios that differ only in the order of supertypes, such as scenario 7a and scenario 7b. Syntactically, this is accomplished by varying the lexical order of the *implementations*. We have defined a semantics that is symmetric with respect to order, so the results are the same for both scenarios. In some languages with multiple code inheritance, such as CLOS [3], the order is significant. In our semantics, the order is not significant. However, the order-dependent sub-scenarios are included in Figure 6 so that the interested reader can trace the JVM modifications described in Section 8 to confirm that our algorithm produces symmetric inheritance semantics.

Note that when a method `alpha()` appears in a superclass, that superclass may actually represent the class `Object`. For example, scenario 5 can be used to illustrate the situation where class `A` represents the `Object` class and the `alpha()` method represents the `toString()` method.

For the scenarios in Figure 6, consider a call-site where the static type of the receiver is any type (*implementation* or class) shown in the scenario, the dynamic type of the receiver is class `C`, and the method signature is `alpha()`.

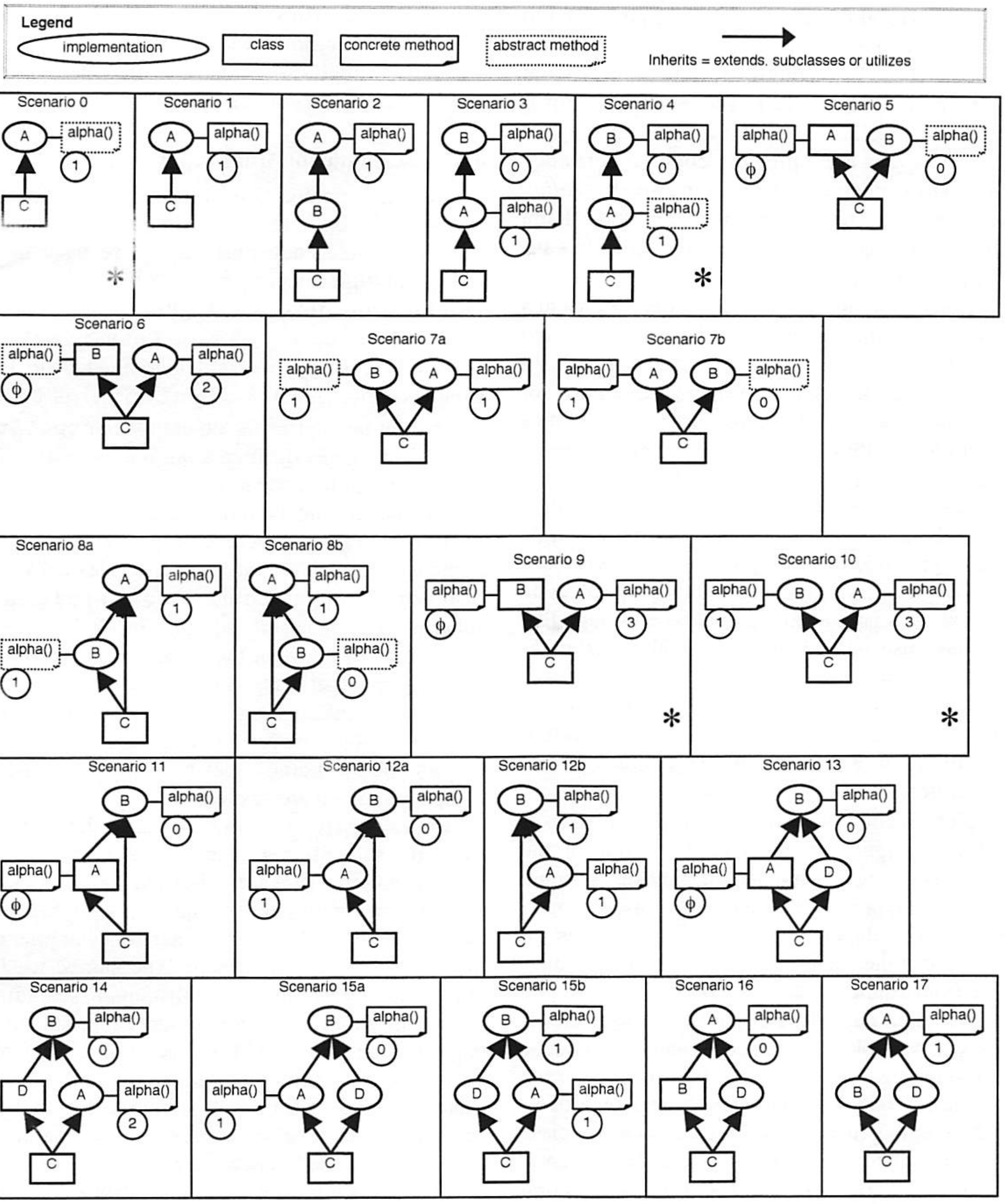


Figure 6. Inheritance scenarios for multiple code inheritance. The circles are explained in Section 8.

Scenario 0 mirrors the traditional case, where an abstract method (no code) in an interface or class is inherited in class C. Since our scenarios assume that the receiver is an instance of class C, some of the scenarios actually produce compiler errors. Such scenarios are marked with an asterisk (*) in the lower right corner. For example, scenario 0 would produce a compiler error, since it would force class C to be abstract and generate an error when the code tries to create an instance of class C. However, it is important to support such scenarios in the JVM with the correct semantics, since these scenarios can occur at runtime, if classes are recompiled in a specific order.

For example, scenario 0 can occur if *implementation* A is compiled with a non-abstract `alpha()` method, then class C is compiled and then *implementation* A is recompiled after changing `alpha()` to be abstract. If class C is not recompiled (legal in Java), then the JVM must throw an exception indicating that the code tried to execute an abstract method.

For scenarios 1 through 3, the code from *implementation* A is dispatched. The semantics mirror the single code inheritance semantics used by classes. Scenario 4 is an example of code inheritance suspension, where an abstract method in *implementation* A blocks class C from inheriting the code from *implementation* B. Scenario 4 mirrors the semantics for code inheritance from classes.

In scenarios 5 through 8, class C inherits code from a class or *implementation* along one inheritance path and an abstract method (no code) along a second path. In this case we define the code inheritance semantics for class C to inherit the code (in particular from A). Notice that scenario 5 directly mirrors the classic case where a class inherits code for a method from a superclass and implements an interface containing an abstract method with identical signature. Scenarios 8a and 8b illustrate an important principle of code inheritance suspension – an abstract method in a type can only suspend code inheritance along a path from a parent type through that type; it cannot suspend code inheritance along *all paths* from its parent type.

In each of scenarios 9 and 10, a *multiple code inheritance ambiguity* exists between two different implementations of `alpha()` in two parent types of class C. Therefore, the programmer is required to supply a local implementation of method `alpha()` in class C to clear this ambiguity. If the method in one of the parent types is desired, a method that makes a single super call to the appropriate parent can be used. Again, the order of inheritance is ignored and there is no preference for inheritance from a superclass over inheritance from a *superimplementation*.

Scenarios 11 through 15 are quite interesting cases. Two different multiple inheritance semantics could be defined for our language extension [19]. *Strong multiple code inheritance semantics* requires these scenarios to be inheritance ambiguities, since for each scenario, class C can inherit different code along different code inheritance paths. However, *relaxed multiple code inheritance semantics* states that if two types serve as potentially ambiguous code sources and are related by an inheritance relationship, the code in the child type overrides the code in the parent type. The code in the child type is called the *most specific method*. With these semantics, the inherited code in class C is the code provided by type A, for scenarios 11 through 15. We have implemented relaxed multiple code inheritance semantics in our compiler and JVM. It would be simple to implement strong semantics instead. In this case, a further decision would be required to resolve scenarios 16 and 17, since they inherit the *same code* along multiple paths. However, since we used the relaxed definition of multiple code inheritance semantics, these are not ambiguous scenarios and the code from *implementation* A is inherited by class C.

4.2 Semantics of Multi-super Calls

Even if a method is overridden, it is often desirable to invoke the original method in a supertype. However, due to multiple code inheritance, we have to choose among multiple supertypes in a super call. We saw an example of this in Figure 4. The method `writeChar(int)` in class `DataOutputStream` needed to call the overridden method code from its *superimplementation* `OutputCode`, as opposed to calling the method in its superclass, `OutputStream`. We call this generalization a *multi-super call*. We refer to a super call to a superclass as a *classic* super call to differentiate it from a multi-super call.

In C++, each multi-super call is a direct jump to a particular superclass that is specified lexically and it is resolved at compile-time (*e.g.* `A::alpha()`). If subsequent changes to the code result in a new class being inserted between the class that contains the call-site and the target class of the multi-super call, then any code in the intervening classes is ignored. This can result in a logic error if one or more of the inserted classes adds a method that performs some additional computations that are desired.

In the spirit of Java, we have defined a more dynamic semantics for multi-super than the static semantics defined for C++. For example, Java does not force the recompilation of sub-classes when a super-class is recompiled. Our modified Java compiler ensures that only a direct parent supertype can be used in the multi-super call. These multi-super semantics are consistent with Java's classic super mechanism, where the lookup always starts from the closest superclass and searches upwards for appropriate method code. If a new superclass is added, the call-site code does not need to be changed to take advantage of any "value-added" code that is inserted in new intervening classes.

Figure 7 shows the basic inheritance scenarios that define the semantics of multi-super. In each scenario, assume that a class or *implementation* (not shown in the scenario) is a direct subtype of *implementation* C and makes a multi-super call to *implementation* C. The scenarios in Figure 7 can be derived from the scenarios of Figure 6 by changing class C into *implementation* C. However, several of the scenarios have been excluded after this transformation, since it is impossible to have a class that is a supertype of an *implementation*. Therefore, scenarios 5, 6, 9, 11, 13, 14 and 16 from Figure 6 are excluded.

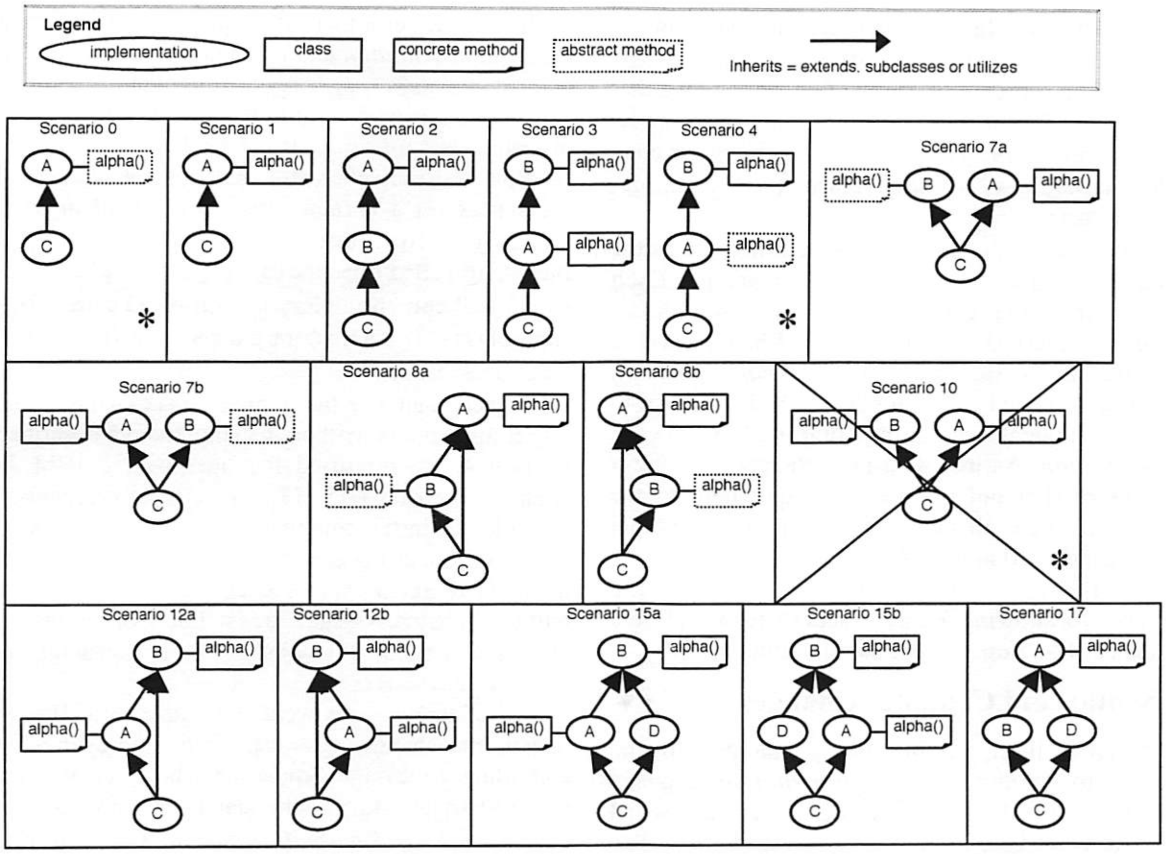


Figure 7. Inheritance scenarios for multi-super.

In addition, scenario 10 cannot occur, since an ambiguous method exception would be generated when *implementation* C was loaded. In other words, ambiguous super calls can never happen. In each scenario of Figure 7, the code for method `alpha()` in *implementation* A is executed. Of course, if this `alpha()` is abstract, then an exception is thrown. As in Figure 6, any scenario marked with an asterisk will not compile without an error, so a series of recompilations is necessary to generate the scenario at runtime.

Note that Figure 7 does not contain any scenarios where a classic super call is made to a superclass, even though the code in this superclass may actually be inherited from an *implementation*. In fact, there are many such scenarios, since each scenario in Figure 7 could have C as a class instead of an *implementation*. In all of these cases, the same semantics hold as if C is an *implementation*.

4.3 Classic Super Calls in *Implementations*

It does not make sense to have a classic super call in an *implementation*, since an *implementation* cannot have a superclass. However, if the same method appears in two classes that share a *superimplementation* and if that method contains a classic super call, promotion of this method to the *superimplementation* would result in a classic super call in this *superimplementation*. For example, assume that there is a common method in `DataInputStream` and `RandomAccessFile` of Figure 5, and that this common method contains a classic super call. The super call should invoke code in `InputStream` if the common method is invoked on an instance of `DataInputStream` and should invoke code in the `Object` if the common code is invoked on an instance of `RandomAccessFile`. However, it is illegal to put a classic super call in an *implementation*, since an *implementation* cannot have a superclass. There are three solutions to this problem in MCI-Java.

If the programmer wants to promote a method to an *implementation* that contains a classic super call, the classic super call should be replaced by a call to a new method. For example, a call `super.alpha()` should be replaced by a call, `this.superalpha()`. The new method should be implemented in each of the subclasses to contain a single line classic super call to the original method. For example, the single line in `superalpha()` would be `super.alpha()`.

A second solution is to modify MCI-Java so that including a classic super call in an *implementation* is legal and the semantics are defined as follows. At runtime, when a classic super call is made in an *implementation* (for example, InputCode), the JVM looks down the calling stack to the first stack frame that is a class, rather than an *implementation* (for example, DataInputStream) and then starts looking for code in the superclass of this class (for example, InputStream).

The third solution defines the same semantics as the second, but uses a different approach. Each *implementation* that contains a method with a classic super call is marked as it is loaded. When a class is loaded that inherits from such a marked *implementation*, the method is treated as though the method was local to the class, instead of being inherited from the *implementation*. As indicated in Section 9, copying a code pointer (but not the code) is equivalent to the approach taken for all methods (not just methods that contain a super call) in *traits* [21].

In MCI-Java, an *implementation* cannot have a superclass, so allowing a classic super call would be a poor choice. Therefore, we use the first solution.

5 Syntax and Compiler Changes

We made three minor syntax changes to the language to support the *implementation* language construct. First, we added the keyword implementation to mark an *implementation*. For example, the first line of the OutputCode *implementation* shown in Figure 5 is:

```
public implementation OutputCode implements
   DataOutput, Sink {
```

Second, we added the keyword utilizes to mark a class that inherits from an implementation. For example, the first line of the RandomAccessFile class shown in Figure 5 is:

```
public class RandomAccessFile utilizes
   InputCode, OutputCode {
```

and the first line of the DataOutputStream class shown in Figure 5 is:

```
class DataOutputStream extends
   OutputStream utilizes OutputCode {
```

Third, we modified the syntax of the super method call to implement the multi-super call. We specify one of many potential *implementations* that can contain code or inherit code from other *implementations*, as an argument. For example, Figure 4 shows a multi-super call from the method writeChar(int) in class DataOutputStream to the *superimplementation* of this method in *implementation* OutputCode.

If no code for a method is contained in a referenced *superimplementation* and it has not inherited code from one of its *superimplementations*, then the compiler generates an error, similar to the case of finding no code in a superclass for a normal super call.

Note that each class still has a unique superclass, so the syntax for a normal super call is unchanged. For example, for any method in the class DataOuputStream, the call super.alpha() would still call an implementation of alpha() in the superclass of DataOutputStream, which is OutputStream.

To accommodate these three syntax changes, and to report ambiguous method declarations, as described in Section 4, we modified the open-source IBM Jikes compiler, version 1.15 [13]. There were originally 243 C++ classes in the compiler. We modified 17 methods in 10 of these classes and added an additional 41 methods to them. We also added 2 classes used in building abstract syntax trees. Each of these added classes consisted of 1 constructor, 1 destructor and 3 accessor functions.

All of the changes were straightforward. However, one of the changes was especially interesting: when compiling a message expression whose receiver is the pseudo-variable, this, the standard compiler always generates an *invokevirtual* instruction. However, if such a message expression appears in an *implementation*, it must generate an *invokeinterface* instruction instead.

6 Correctness and Performance Experiments

This Section provides an overview of tests and experiments conducted during the process of verifying our modifications to the Jikes compiler and SUN JVM JDK 1.2.2. The first goal of our validation was to show that our multiple code inheritance implementation preserves the semantics and performance of existing single inheritance code. The second goal was to show that both our basic multiple code inheritance and the multi-super call mechanism execute correctly in multiple inheritance programs.

We first compiled and ran three large existing Java programs (javac, jasper and javap) using our modified compiler and JVM. In all three of these tests, we obtained correct results and there was no measurable change in the execution times, between the original and modified JVMs [8].

We then conducted tests to verify the correctness of our JVM and compiler modifications for multiple inheritance programs. We constructed test programs for each scenario described in Section 4 and they produced the desired results. The scenarios shown in Figure 6 test all paths through the modified class loader code shown in Figure 8. The scenarios in Figure 7 test all paths through the modified multi-super resolution code.

Finally, we conducted an experiment to evaluate the runtime performance of the refactored I/O classes described in Section 2 that used multiple inheritance, compared to the I/O classes from the standard library. These refactored library classes exercise all of the modifications that we made to support multiple code inheritance, including the use of the pseudo-variable `this` in an *implementation*. The test program ran without errors and with unmeasurable time penalties for multiple code inheritance. We used two different configurations. The first used an AMD Athlon XP 2400+ running Red Hat linux version 7.2. The second used a SUN Ultra-60 running Solaris version 9.

This test program starts by creating an instance of `RandomAccessFile` and writing a series of double precision values, int values, char values and strings to it. This exercises methods specified in the interface `DataOutput` whose code appears in the *implementation* `OutputCode`. The data file is then closed and reopened as an instance of `DataInputStream`. All of the data is read, using methods specified in the interface `DataInput`, whose code appears in the *implementation* `InputCode`, with help from a few methods that remain in `DataInputStream`. As the data is read, it is written to a second file using an instance of `DataOuputStream`. These writes exercise methods specified in `DataOutput` and implemented in `OuputCode`. Finally, this file is read using an instance of `RandomAccessFile`, exercising methods specified in `DataInput` and implemented in `InputCode`.

7 Dispatch in the Unmodified JVM

To implement multiple code inheritance, we modified SUN's JVM 1.2.2 [23] to execute code in interfaces. We know of no elegant way to implement multiple code inheritance in Java without JVM modifications. Although the approach of using inner classes [18] is interesting, its use of delegation inheritance along the interface chains is not very appealing from the language consistency perspective. Inner classes make interface inheritance second class. We have previous success in modifying JVM dispatch to support multi-method dispatch [10]. Our changes to support multiple code inheritance are concise and localized and should transfer to other JVMs.

In this Section we briefly review how a method call-site is dispatched in the unmodified SUN JVM and the standard data structures that are used. A more complete description of these data structures has appeared [8].

At compile-time, a call-site is translated to a JVM instruction whose bytecodes depend on the static type of the receiver object. If the static type is a class, then the generated opcode is *invokevirtual*. If the static type is an interface, then the opcode is *invokeinterface*. In either case, a method reference is also stored as an instruction operand – an index into the constant pool. The method reference contains the signature of the method and the static type of the receiver object.

In the SUN JVM, the dispatch of *invokevirtual* uses three data structures: *method block* (MB), *method table* (MT) and *virtual method table* (VMT). The dispatch process for *invokeinterface* requires one additional data structure, *interface method table* (IMT).

At runtime, the compiled code for each method is referenced using a *method block* (MB) that contains complete information for the method, including its signature, a pointer to its bytecodes and an offset that is used during dispatch. For interfaces, the bytecode pointer is `null`, since in standard Java there can be no code in interfaces. In the SUN JVM 1.2.2 distribution, method dispatch consists of the following three steps:

S1. Method resolution: generates a *resolution method block*.

S2. Method quicking (or pre-execution): replaces the opcode with one of its quick counterparts and computes a reference to an *execution method block*.

S3. Method execution: executes the quicked bytecode using the referenced execution method block.

A *method table* (MT) is an array of MBs that are declared (not inherited) in a class or interface. To resolve an *invokevirtual* instruction (S1), the JVM uses the bytecode's method reference to obtain the static class and a method signature. It then searches the MT of this static class for an MB whose signature matches. If no match is found, it searches the MTs along the superclass chain. The compiler guarantees that a match is found and the match is the resolution MB.

The resolved MB will not necessarily be executed. However, it will contain an offset that can be used as an index into another data structure called the *virtual method table* (VMT), which contains a pointer to the execution MB. A reference to this execution MB is used in the quick bytecodes that are generated in step 2 (S2) of the method dispatch process. We make no modifications to steps S2 or S3.

When a class is loaded, the loader constructs an MT and VMT for the class. It constructs a VMT by first copying the VMT of its superclass and then extending the VMT for any new methods that have been declared in the class, whose signatures are different from the signatures of inherited methods. During class loading, the loader may discover that the class needs a VMT slot for an abstract method for which it does not declare any code or inherit any code. In this case, the loader extends the VMT by providing a slot for this method, allocates an MB in another table called the *Miranda Method Table* (MMT) and sets the VMT slot for the method to point to this new MB. At the end of this process, every method that can be invoked on an instance of the loaded class has a unique VMT table entry that points to an MB. We refer to a VMT entry that points to an MB in the class's MT or MMT as a *local VMT entry*. It is also possible that a VMT entry points to an MB in the MT or MMT of a superclass. Such an entry is called a *non-local VMT entry*. This distinction is critical to support code in interfaces.

Resolution of *invokeinterface* (S1) is similar to resolution of *invokevirtual*, except the method reference uses an interface instead of a class. Resolution starts at the *interface method table* (IMT) of the interface.

The IMT provides an extra level of indirection that solves the problem of inconsistent indexing of interface methods between classes. This extra level of indirection is analogous to the way C++ implements multiple inheritance using multiple virtual function tables.

An IMT has one entry for each interface that is extended or implemented (directly or indirectly) by its class or interface. This entry contains a pointer to the interface and a pointer to an array of VMT offsets, where there is one array entry for each method declared in the interface.

During resolution, the JVM starts with the zero'th entry of the interface's IMT, which contains a pointer to the interface itself. The MT of this interface is searched for a matching method. If one is not found, the MTs of subsequent interfaces in the IMT are searched. The compiler guarantees a signature match.

As with the *invokevirtual* bytecode, the resolution MB may not be the execution MB. In the *invokeinterface* case, the resolution MB contains a local MT offset instead of a VMT index. To use this offset, the JVM first finds the IMT entry for the static interface type of the receiver object. This entry contains an array of VMT indexes. The offset from the resolution MB selects the array element that contains the appropriate index into the VMT of the receiver's class. This VMT entry is the execution MB and a reference to it is used in the quick bytecodes that are generated in step 2 (S2) of the dispatch process. A good description of alternate approaches to implementing *invokeinterface* , including class object search, itable search, indexed itables and the alternate IMT scheme used in the Jikes RVM (formerly Jalapeño) has appeared [2].

8 JVM Modifications for Multiple Code Inheritance

In this Section we describe the localized changes we made to the SUN JVM to support multiple code inheritance. Recall that at the source code level, we support multiple code inheritance by placing code into a new construct called an *implementation*. However, our compiler produces a .class file that represents each *implementation* by an interface with method code. Therefore, our changes to the JVM are based on supporting code in interface .class files.

8.1 Code in Interfaces

Since code from interfaces has to be reachable from the class pointer of the receiver object, we modify the IMT construction for a class to copy the code from the interfaces to a data structure accessible from the class. Since this change only affects JVM class loading code and does not change any code that is executed at a call-site, its runtime overhead is small.

In the current JVM, when constructing the IMT of the loaded class, the JVM iterates over each superinterface of the class. For each interface, the JVM iterates over each declared method. Besides the normal actions taken in the classic JVM, for each method in an interface, our modified JVM takes some additional actions. The algorithm that implements these extra actions is shown in Figure 8. Each scenario from Figure 6 is marked in the algorithm to show where it is handled. Each method in Figure 6 has a circled number that shows which action from Figure 8 is taken when its MB is processed. The circled ϕ indicates that no action is taken, since that method is in a class instead of an interface. In the algorithm, the symbol < is used to indicate a proper subtype and >= indicates a supertype.

Creating a new MB on the C-heap is necessary when a class inherits code from an interface that overrides code in a non-local MB. The alternative of changing the code pointer for a non-local MB can result in the wrong code being executed. For example, consider scenario 14 from Figure 6. At the time when class `C` is being loaded, its VMT entry for `alpha()` points to a non-local MB in the MMT of class `D`. While building the IMT in class `C`, the JVM encounters the method `alpha()` in interface `A`. If it copied the MB for `alpha()` from interface `A` to the MB for `alpha()` in class `C` (stored in the MMT of class `D`), dispatch would work properly for any `alpha()` message sent to an instance of class `C`. However, consider the message `alpha()` sent to an instance of class `D`. Its VMT entry for `alpha()` points to the modified MB in its MMT, which points to the code in interface `A` (instead of the code from interface `B`).

```
Let c be the class being loaded.
Let imb = the MB of the method being processed.
Let mb = the current MB pointed to by the VMT
   entry of c with the same signature as the
   method being processed.
if (mb.codepointer == null)
   if(mb is not local(c))
      Action 2 //scenarios: 6A
   else if (imb.type > mb.type) and there is no
      path from c.type to imb.type that does
      not go through mb.type
      Action 0 // scenarios: 4B
   else
      Action 1 // scenarios: 0, 1, 2A, 3A,4A,
               // 7aB, 7aA, 7bA, 8aB, 8aA, 8bA,
               // 10B, 12aA, 12bB, 15aA, 15bB
               // 17A
else // mb.codepointer != null
   if (imb.codepointer == null)
      Action 0 // scenarios: 5B, 7bB, 8bB
   else
      if (imb.type < mb.type)
         if (mb is local(C))
               Action 1 // scenarios:
                        // 12bA, 15bA
         else
               Action2 // scenarios: 14A
      else if (imb.type >= mb.type)
         Action0 // scenarios 3B, 11B, 12aB,
                 // 13B, 14B, 15aB, 16A,
      else // imb.type unrelated to mb.type
         Action 3 // scenarios: 9A,10A

Action 0: do nothing.
Action 1: imb is copied onto mb, but the offset
of mb (index back to the VMT) is retained.
Action 2: Create a new MB on the JVM C-heap,
copy imb to the new MB, change the current VMT
entry (the index of this entry is in the offset
of mb) to point to the new MB and change the
offset of the new MB to this VMT index.
Action 3: Throw an ambiguous method exception.
```

Figure 8. The JVM modifications to support code in interfaces.

It is important to note that resolution and dispatch of *invokevirtual* and *invokeinterface* bytecodes proceed in exactly the same way as with the unmodified JVM, but the change in the class loading code allows the code in the interface to be found and executed. With the design choices we made, no other JVM changes were required to support code in interfaces. That is, we modified the IMT construction algorithm for a class to:

1. detect and report potential ambiguities, and
2. copy the code from interfaces to classes.

8.2 Multi-super Calls

To support multi-super calls, we changed the resolution code that is executed the first time a multi-super call-site is encountered. There are no changes to the dispatch code, so any multi-super call-site that is executed more than once uses the standard JVM code for subsequent executions. For example, the JVM actually has an assembly language module that does dispatch (instead of using C code). No change to this assembly language dispatch module is required to support our multi-super extension.

A super call is compiled into an *invokespecial* bytecode, where the method reference contains a target interface instead of a class. The JVM can recognize a multi-super call, since it is the only case where the compiler generates an *invokespecial* bytecode and the method reference is an interface instead of a class.

Our modified JVM uses a custom resolution algorithm to find a resolution MB. The algorithm traverses all of the interfaces in the IMT of the target interface. It finds all MBs whose signature matches the signature in the method reference. It then computes the most-specific MB as the resolution MB. Our JVM then finishes by performing the same bytecode quicking operation as the unmodified JVM, where the call-site is replaced by an *invokenonvirtualquick* bytecode. No change is made to the way this quicked bytecode is executed (in the assembly language module).

Since resolution happens only once at each call-site, the overhead of our change is insignificant in the running time of a program, as supported by the performance experiment described in Section 6.

9 Related Work

In Section 1, we surveyed the related work in multiple inheritance on a per-language-basis. Researchers, including our group, have prototyped a variety of new approaches to multiple inheritance without the new features ever being a standard part of a language. In terms of programming language concepts, the most-closely related work to MCI-Java's *implementations* are *mixins* [1] and *traits* [21].

The main differences between the three approaches are (1) the base programming language used to prototype the feature, (2) compiler support, (3) the semantics of the inheritance, and (4) the semantics of dynamic code handling (e.g., compile-time versus load-time versus runtime). The most important differences are related to semantics, but the details of the base language and compilation are also significant. For example, since MCI-Java is implemented within a full-fledged Java VM, our work has had to solve a number of practical issues related to dynamic code, where the source code for the classes is not available to the VM. In contrast, *traits* have been prototyped within Smalltalk, which allows for all affected code to be recompiled when necessary.

The *traits* paper itself has an excellent overview of *mixins*, including a good description of how they address multiple inheritance [21]. *Mixins* have been investigated using Java as well, but *mixins* provide compositional inheritance semantics, as opposed to MCI-Java's hierarchical inheritance semantics. Consequently, *mixins* have three main problems: ordering, dispersal of glue code, and fragile hierarchies. In the interests of space, we do not repeat all of the observations made in the *traits* paper here. However, to summarize: in MCI-Java, inheritance is symmetric so ordering does not apply. There is no glue code in MCI-Java, so this is not a problem. MCI-Java solves most (but not all) of the problems with fragile hierarchies. However, MCI-Java always finds the appropriate method to use at runtime by not copying any method pointers at compile-time. In addition, all errors due to hierarchy changes that can be determined at load-time are reported at load-time and errors that cannot be determined at load-time are handled as exceptions at runtime. There are no unexpected executions with MCI-Java.

We now focus on comparing MCI-Java's *implementations* with *traits*. Our work with MCI-Java has evolved over a couple of years to include compiler support for *implementations*, more localized changes to the Java VM, and better handling of dynamic code loading. For example, an earlier version of MCI-Java, reported in Cutumisu's Master's thesis (2002) [8], used scripts to handle Java source language changes instead of a modified Java compiler. In spirit, the goals of *traits* are closely aligned with our language goals, since in addition to solving the code reuse problem, *traits* also provides a separate language component for code that is independent of language features for interface and representation.

There are many similarities between an *implementation* and a *trait*. Each is a collection of methods (code). Each can be used by a client class to augment its natively-defined (locally-defined) methods. Each can invoke methods that are abstract until they are natively defined in a client class that uses it. Each contains no representation information (instance variables). A class can use more than one *implementation* or *trait*, which can lead to inheritance conflicts. The semantics of inheritance conflicts and the resolution mechanisms are different in *traits* and MCI-Java and they will be discussed later. There are also many differences between an *implementation* and a *trait*. However, since there is not enough space to cover all of the differences, we will focus only on the fundamental distinctions.

The most fundamental difference between an *implementation* and a *trait* is that when a client class that uses a *trait* is compiled, the non-overridden methods from the *trait* are *flattened* into the client class. This means that the methods can be viewed as if they were defined natively in the client class. This does not mean that the code is copied to the client class, since the main goal is to allow the code to be shared by different client classes. Sharing is accomplished by extending the Smalltalk method dictionary (similar to a symbolic virtual method table) for each client class that uses a *trait* by one entry for each of the non-overridden methods. However, the entries in all method dictionaries that use a *trait* point to common code stored in the *trait* itself, where the *trait* is a "hidden class".

The extension of the method dictionary occurs when the client class is compiled and this is the source of many of the most important differences between *implementations* and *traits*. In Smalltalk, if a superclass or a *trait* used by a client class is recompiled, the client class is automatically recompiled. This can be done, since all of the classes and *traits* are in the same Smalltalk image. In Java, if a superclass or an *implementation* used by a client class is recompiled, the client class is not automatically recompiled. In Java, a class decides at load-time (not compile-time) which methods it will put in its method table. As a simple example, consider scenario 2 from Figure 6. Assume `A` and `B` are both *implementations* and scenario 2 applies when class `C` is compiled. However, assume that before runtime, *implementation* `B` is recompiled so that it contains a method for `alpha()`. Even though class `C` is not recompiled, the correct code in *implementation* `B` is executed, since the method table for class `C` does not contain any methods (or pointers to methods) that were copied to it when it was compiled.

This example also applies if `B` and `C` are classes. In fact, *implementations* were designed to behave as classes in this respect and mirror the semantics of Java. This is just one example of why changing method tables at compile-time (i.e., flattening) and hiding the code source at runtime is problematic in Java. *Implementations* survive at runtime as first-class language features, not as hidden entities that serve only as repositories for shared code. Again, this is the most fundamental distinction between *traits* and *implementations*.

More generally, the *traits* paper [21] has an excellent description of the three major problems with multiple inheritance: "conflicting features", "accessing overridden features" and "factoring out generic wrappers". Fortunately, MCI-Java has solved all three of the problems.

As indicated in the *traits* paper, the "conflicting features" problem is not really a problem if data is not multiply-inherited and multiple-data inheritance is disallowed in MCI-Java. One difference between *traits* and MCI-Java is the conflict resolution semantics. MCI-Java and *traits* both solve the "diamond" problem (scenario 17 of Figure 6) by declaring no conflict. However, MCI-Java adopts a more relaxed definition of inheritance conflict [19] than *traits*. Consequently, scenarios 12a, 12b, 14, 15a and 15b, which are not inheritance conflicts in MCI-Java, would be inheritance conflicts with *traits*, if the *implementations* were replaced by *traits* and no glue was used. The conflict resolution can also result in different methods being selected by MCI-Java and *traits*. If the *implementations* in scenarios 11 and 13 were replaced by *traits*, there would still be no inheritance conflicts, but *traits* would select the method in `B` instead of `A` for each case. This is because methods from *traits* take precedence over methods from superclasses, whereas in MCI-Java, inheritance from superclasses and inheritance from *superimplementations* are treated the same.

It is also correctly pointed out in the *traits* paper that explicitly naming an arbitrary superclass in the source code (as done in C++) makes the code fragile with respect to changes in the architecture of the class hierarchy. That is why MCI-Java requires every explicit multi-super call to be made to a direct *superimplementation* and inheritance is then used to find the appropriate method. For example, in scenario 17, the programmer must decide whether a super call in *implementation* (or class) `C` should be along the `B` inheritance chain or `D` inheritance chain, rather than explicitly specifying *implementation* `A`. If the inheritance hierarchy is changed above the direct superclasses, the original intent to inherit along the `B` or `D` inheritance chain will survive. Note that radical changes to the inheritance hierarchy above the immediate superclasses can be made in Java without recompiling type `C` and no unexpected consequences will arise. However, the only way to change the inheritance relationships between type `C` and its immediate *superimplementations* is to recompile type `C`. In this case, if the original super call is made invalid due to a direct *superimplementation* being removed, the compiler will generate an error, so no unexpected consequences will arise.

The *traits* paper also has a pertinent example of the third problem with multiple inheritance: factoring out generic wrappers. However, once again, this problem has been solved in MCI-Java. The problem can be reduced to solving the problem of what to do with classic super calls that get promoted to *implementations* as described in Section 0.

In summary, MCI-Java addresses almost all of the weaknesses attributed to *mixins* and, more generally, various aspects of multiple inheritance [21]. Although there are many similarities between MCI-Java and *traits*, the fundamental difference is the choice, for *traits*, of inheritance semantics based on flattening. The compile-time-based technique of flattening is difficult to support in Java, given the dynamic semantics of the VM and the inaccessibility of source code at load-time and runtime. The dynamic aspects of Java led to many scenarios and implementation problems (e.g., Sections 4 and 8) that we solved for MCI-Java, which are not issues for *traits* under Smalltalk.

10 Conclusions

We have shown why multiple code inheritance is desirable in Java. We have defined a mechanism, called *implementations*, which supports multiple code inheritance and a super call mechanism that allows programmers to specify an inheritance path to the desired *superimplementation*. We have defined simple syntactic Java language extensions and constructed a modified Jikes compiler (*mcijavac*) to support these extensions. We have constructed a modified JVM (*mcijava*) to support multiple code inheritance. Our modifications are small and localized. The changes consist of:

1. Class loader changes to support code in interfaces.
2. Method block resolution changes to support multi-super.

Our JVM modifications do not affect the running time of standard Java programs and they add negligible overhead to programs that use multiple inheritance.

11 Acknowledgement

This research was supported by grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank the referees and the programming committee for several helpful suggestions that improved the manuscript. Finally, we would like to thank the shepherd for this paper, Michal Cierniak, for his help.

References

[1] D. Ancona, G. Lagorio and E. Zucca, Jam – A Smooth Extension of Java with Mixins, *14th European Conference on Object-Oriented Programming (ECOOP)*, Cannes France, pp 145-178, June 2000.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove and D. Lieber, Efficient Implementation of Java Interfaces: InvokeInterface Considered Harmless, *16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA*, Tampa U.S.A., pp 108–124, October 2001.

[3] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon, Common Lisp Object System Specification, X3J13 Document 88-002R, June 1988.

[4] E.L. Burd, M. Munro, Investigating the Maintenance Implications of the Replication of Code, *Proceedings of the International Conference on Software Maintenance (ICSM)*, Bari Italy, pp 322 – 329, October 1997.

[5] C. Chambers and G. T. Leavens. BeCecil, A Core Object-Oriented Language With Block Structure and Multimethods: Semantics and Typing, *Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Paris France, January 1997.

[6] C. Chambers and G. T. Leavens, Typechecking and Modules for Multimethods. *A C M Transactions on Programming Languages and Systems*, 17(6), pp 805–843, November 1995.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, An Empirical Study of Operating Systems Errors. *18th ACM Symposium on Operating System Principles (SOSP)*, Banff Canada, pp 73–88, October 2001.

[8] M. Cutumisu, Multiple Code Inheritance in Java, M.Sc. Thesis, University of Alberta, http://www.cs.ualberta.ca/~systems/mci/thesis.pdf, December 2002.

[9] M. Day, R. Gruber, B. Liskov, and A. C. Myers, Subtypes vs Where Clauses: Constraining Parametric Polymorphism, *10th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA*, Austin U.S.A., pp 156–168, October 1995.

[10] C. Dutchyn, P. Lu, D. Szafron, S. Bromling and W. Holst, Multi-Dispatch in the Java Virtual Machine: Design and Implementation, *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio U.S.A., pp 77-92, January 2001.

[11] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, New Jersey, 1990.

[12] M. Franz, The Programming Language Lagoona — A Fresh Look at Object-Orientation, *Software — Concepts and Tools*, 18, pp 14–26, 1997.

[13] IBM Research Jikes Compiler Project http://www.ibm.com/developerworks/opensource/jikes/

[14] Y. Leontiev, M. T. Özsu, and D. Szafron, On Separation between Interface, Implementation and Representation in Object DBMSs, *26th Technology of Object-Oriented Languages and Systems Conference (TOOLS USA)*, Santa Barbara U.S.A., pp 155 – 167 August 1998.

[15] Y. Leontiev, T. M. Özsu and D. Szafron, On Type Systems for Database Programming Languages. *ACM Computing Surveys*, 34(4) pp 409 – 449 December 2002.

[16] MCI-Java. http://www.cs.ualberta.ca/~systems/mci.

[17] B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice-Hall, 1997.

[18] M. Mohnen, Interfaces with Skeletal Implementations in Java, *14th European Conference on Object-Oriented Programming (ECOOP 2000) - Poster Session*, June 12th - 16th 2000, Cannes, France, http://www-i2.informatik.rwth-aachen.de/~mohnen/PUBLICATIONS/ecoop00poster.html has a link to an unpublished full paper.

[19] C. Pang, W. Holst, Y. Leontiev and D. Szafron, Multi-Method Dispatch Using Multiple Row Displacement, *13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon Portugal, 304-328, June 1999.

[20] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, Emerald: A General-Purpose Programming Language. *Software Practice and Experience*, 21(1), pp 91–118, January 1991.

[21] N. Schärli, S. Ducasse, O. Nierstrasz and A. Black, Traits: Composable Units of Behavior, *17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt Germany, pp 248-274, July 2003.

[22] D. Stoutamire, and S. Omohundro, The Sather 1.1 specification. Tech. Rep. TR-96-012, International Computer Science Institute, Berkeley, August 1996.

[23] Sun Microsystems Inc. Java[tm] 2 Platform. http://www.sun.com/software/communitysource/java2/download.html.

Semantic Remote Attestation — A Virtual Machine directed approach to Trusted Computing

Vivek Haldar, Deepak Chandra and Michael Franz
Department of Computer Science
University of California
Irvine, CA 92697-3425
{vhaldar,dchandra,franz}@uci.edu

Abstract

Remote attestation is one of the core functionalities provided by trusted computing platforms. It holds the promise of enabling a variety of novel applications. However, current techniques for remote attestation are static, inexpressive and fundamentally incompatible with today's heterogeneous distributed computing environments and commodity open systems. Using language-based virtual machines enables the remote attestation of complex, dynamic, and high-level program properties — in a platform-independent way. We call this *semantic remote attestation*. This enables a number of novel applications that distribute trust dynamically. We have implemented a prototype framework for semantic remote attestation, and present two example applications built on it — a peer-to-peer network protocol, and a distributed computing application.

1 Introduction

Two major trends have been immensely influential in today's computing environment. The first is heterogeneity. We compute using everything from small cellphones, to handhelds, to desktop workstations, to rack-mounted servers — and these must all inter-operate. This has led to the widespread acceptance of open protocols for communication and portable language runtimes (such as the Java and .NET virtual machines) for execution of programs.

The second major trend is mobility. Not only must all these varied computing devices inter-operate seamlessly, we must also be able to use most of our familiar programs and data across all of them. Both of these have significantly increased the importance of having a so-called "common platform". This common platform is increasingly the language runtime, that executes some form of platform-independent mobile code.

As we become dependent on this computing infrastructure, its weaknesses also become painfully apparent. On the one hand, we have periodic waves of network outages caused by worms such as Nimda and SQL Slammer. On the other, content producers want control over the proliferation of their creations. The network is a hostile place — the default assumption is to assume an adversarial remote machine.

One way to get security assurances is to use *closed systems*. They enforce compliance with a certain security policy by being tightly controlled. They are usually manufactured by a single vendor to rigid specifications. Designers have complete control over the whole system and build it specifically to conform to a given security policy. When one closed system communicates with another, it knows within very tight bounds the expected behavior of the remote party. Common examples of closed systems are automated teller machines (ATMs) and proprietary game consoles

Open systems, on the other hand, have no central arbiter. Commodity personal computers and handhelds are examples of open systems. An open system can be easily changed to behave maliciously towards other systems that communicate with it. Two communicating open systems cannot assume anything about each others' behavior, and must be conservative in their assumptions.

Trusted computing is an effort to bring some of the properties of closed, proprietary systems to open, commodity systems. Trusted computing introduces mechanisms and components in both hardware and software that check and enforce the integrity of a system, and allow it to authenticate itself to remote systems. For example, a trusted booting procedure makes sure that the operating system has not been tampered with. Using a chain of reasoning that starts from a trusted hardware module, we can arrive at a conclusion about the state of a system after boot-up. Similarly, we can deduce for sure what particular program is running on a system. *Remote attestation* is the process by which software authenticates itself to remote parties. This allows the remote party to make certain assumptions about the behavior of the software.

Before trusted computing can reach its full potential, questions such as the following need to be addressed:

- how do programs running on trusted platforms authenticate each other in a manner that ensures that each party satisfies some security criteria, while leaving room for various differing implementations?
- the current client-server network computing model assumes a trusted server, and untrusted (even malicious) clients. Thus, even though a significant fraction of work is done at the clients, all the trust resides at the server. How can we design new network protocols (or adapt existing ones) to work in an environment that allows a more flexible partitioning of trust?
- Moving away from the model of having a fully trusted server, and a fully untrusted client, how do we design models and applications that use them, that can broker trust in more flexible and dynamic ways than is possible today?

As we explain in this paper, current methods of remote attestation suffer from many critical drawbacks. It is a technique well-suited to rigidly controlled closed systems, but completely inadequate for the open systems of today, which embody a great variety of hardware and software platforms.

We propose a way out of the problems of standard remote attestation by using a trusted virtual machine as a basis for doing remote attestation. We call this *semantic remote attestation*, since it can certify high-level, fine-grained, and dynamic properties of platform-independent mobile code. The core idea behind semantic remote attestation is to use a *trusted virtual machine* (TrustedVM) to explicitly derive or enforce, on behalf of a remote party, various properties of applications running within it.

Intuitively, "authentication" of an entity should have a broader meaning than it does currently. It should encompass not just verifying the source from which it originated, but also include verifying or proving that its behavior conforms to a required security policy.

To gain experience with semantic remote attestation, we have implemented a prototype TrustedVM, and two example applications on it. The first application is a simplified peer-to-peer networking protocol, and the second is a distributed computing client-server application. Implementing these within our framework achieved two benefits:

- trust relationships between peers, or between clients and servers, were *made explicit, and then checked or enforced* by the TrustedVM. Typically, they are implicit and taken on trust.
- making the trust relationships explicit results in having some knowledge of *degree of trustworthiness* of clients and peers. (for example, knowing which properties were satisfied, and which were not). This allows the applications to make informed decisions about the "goodness" of a result, and dynamically adjust its trust relationships.

The rest of the paper is organized as follows: Section 2 briefly covers basic concepts in trusted computing and remote attestation, and explains the shortcomings of remote attestation as it is today; Section 3 explains how virtual machines can be used for flexible, expressive remote attestation — this forms the core of the paper; Section 4 discusses the implementation issues involved in this and presents two example applications; Section 5 surveys related work; Section 6 discusses avenues for future work; and Section 7 concludes.

2 Background: Trusted Computing and Remote Attestation

The broad goal of trusted computing is to add components and mechanisms to open, commodity systems to bestow on them some of the properties of high-assurance closed systems. This is done using a combination of hardware and software support. It requires three core mechanisms:

- **Secure boot:** to make sure that the system is booted into a trusted operating system that adheres to some given security policy. If such a mechanism is not provided, an adversary could boot the system into a modified operating system that bypasses the security policy.
- **Strong isolation:** to prevent the system from being compromised after it has been booted, and to prevent applications from tampering with each other.
- **Remote attestation:** to certify the authenticity of software being run by a remote party.

The first two guarantee integrity — that the system was not tampered with since it was last turned off (secure boot), and that execution of programs will not be tampered with (strong isolation). The third has to do with authenticity — to be sure of the identity of a remote party or program. The focus of this paper is remote attestation.

At the root of the attestation process is a hardware device called a *trusted module* (TM). This has embedded in it the private key of a public-private key-pair. This key-pair is certified by a certification authority (CA). The TM also has a small amount of non-volatile storage. A hash of the BIOS is signed using the TM's private key and stored here. At boot-up, control is first passed to the TM. It recomputes the hash of the BIOS, and compares it with its stored hash. If they are the same, then we know that the BIOS has not been tampered with, and control is passed to the BIOS. The BIOS does a similar check before passing control to the boot-loader. And the boot-loader does a similar check before passing control to the operating system. All layers along this chain have their own public-private key-pair, which is certified (signed using the private key) by the layer immediately preceding it in the chain. This in turn is used to certify the layer immediately above it. Precisely, each layer signs two things of the layer above it: a hash of its executable image, and its public key. This binds the public key to the software.

Note that the private key of every layer along this chain must be kept secret from the layer immediately succeeding it. Thus, the BIOS must be unable to read the TM's private key, the boot-loader must be unable to read the BIOS's private key and so on. At the successful completion of this chain of checks, the system is guaranteed to have booted into a trusted, untampered operating system.

Thus, a combination of hardware (the trusted module) and software (used for secure booting and isolation) is needed to guarantee both integrity and authenticity of a trusted system. The trusted computing base of this setup consists of the hardware trusted module, and a trusted operating system that handles booting, and enforces strict isolation between applications.

2.1 Remote Attestation — and its problems

Remote attestation is the process by which an application authenticates itself to a remote party. When asked to authenticate itself, an application asks the operating system for an endorsement. The operating system signs a hash of the executable of the application. The entire certificate chain, starting from the trusted module all the way up to the application, is sent to the remote party. The remote party verifies each certificate of this chain, and also checks that the corresponding hashes are of software it approves.

The attestation process must result in the client and server sharing a secret, or else the session can be easily hijacked (e.g. by doing the attestation using one program, and further communication using another).

This method of remote attestation suffers from several critical drawbacks. Briefly, they are:

- It says nothing about program behavior
- It is static, inflexible and inexpressive
- Upgrades and patches to programs are hard to deal with
- It is fundamentally incompatible with a widely varying, heterogeneous computing environment
- Revocation is a problem

We discuss each of these in more detail below.

The most critical shortcoming of remote attestation is that it is not based on program behavior. Even though what is fundamentally sought is some assurance of program behavior (with respect to some security policy), remote attestation certifies something completely different. It simply certifies what exact executable is running. Any assurances about the behavior of the program are taken on trust. It is possible for an attested program to have bugs, or otherwise behave maliciously.

Remote attestation defined in this way is completely static and inflexible. It can convey no dynamic information about the program, such as its runtime state, or the properties of the input it is acting upon. It is a one-time operation done at the beginning of a network protocol.

Another problem is that upgrades and patches are hard to deal with. Linear upgrades from one version to the next can be dealt with by simply updating the list of "approved" software that a remote party uses. In closed and tightly controlled systems such as ATMs, this is tractable.

The situation with widely available commodity software is completely different. As is increasingly common today, upgrades and patches are released very frequently. Also, software is patched more often than it is upgraded. There are usually multiple patches for multiple bugs and insecurities for a given program. Any subset of these patches may be applied in any order. This results in an exponential blowup in the space of possible binaries for a program.

In such a scenario, remote attestation faces problems at both ends of the network. Servers have to manage the growing intractability of maintaining a very large list of "approved software", which is likely to always be behind the current state. Clients, on the other hand, may have to hold off on applying patches or on upgrading, simply to be able to work within a remote attestation framework.

Today's computing ecosystem is extremely varied and accommodates a spectrum of heterogeneous systems with widely varying capabilities. These systems range from high-end supercomputers, to consumer devices such personal computers, handhelds, cellphones and watches, and even ubiquitously embedded microprocessors. Thus, a high premium is placed on portability and interoperability. This is one reason why cross-platform portable solutions such as Java are so popular. Remote attestation, however, with its stress on certifying particular platform-specific binaries, is fundamentally incompatible with this reality. Just as with managing upgraded and patched versions of software, certifying programs that run on a variety of platforms and that must inter-operate with each other, quickly becomes intractable.

Remote attestation inherits a problem from public-key cryptography — revocation. Once a certification authority issues a certificate, it is very hard to revoke. One method is to have publicly available certificate revocation lists (CRLs) which are looked up at regular intervals. Thus, there may be some time lapse between a certificate being revoked, and access being denied to it. Checking with some revocation infrastructure (such as a CRL) at every attestation would be very inefficient.

3 Semantic Remote Attestation

The shortcomings of traditional ways of remote attestation can be traced back to one root cause — *what is desired is attestation of the behavior of software running on a remote machine, but what actually gets attested is the fact that a particular binary is being run.*

Whether a remote party is running, say, Outlook 5.1, is not per se the information that is sought. What is sought is whether that particular program abides by some security policy. However, all that traditional remote attestation is able to certify is simply what exact binary code is running on a remote platform. From this, an indirect implication is drawn about the program's behavior. It is very important to realize that such an assurance gained about a program's behavior is based purely on trust, not on verification. Given the knowledge of what exact program is running on a remote platform, we trust it to behave according to a given policy because, essentially, the vendor of the software claims so. The chain of cryptographic certificates binds this claim to the vendor, and the program.

This is a direct consequence of using purely cryptographic methods for remote attestation. Cryptography is good at certifying entities — it is not suitable for certifying behavior.

The solution we propose leverages the techniques of language-based security and virtual machines. Language-based security [21, 25] has been variously defined as "a set of techniques based on programming language theory and implementation, including semantics, types, optimization and verification, brought to bear on the security question" and "leveraging program analysis and program rewriting to enforce security policies". It derives assurances about the behavior of the code by looking at the code itself.

Recent and very promising examples of this approach include proof-carrying code[24], typed assembly language (TAL), inlined execution monitors[14], and information flow type systems[23]. These techniques can be thought of as falling into two major categories — program rewriting and program analysis. Program analysis covers a variety of techniques that statically try to check a program's conformance to a security policy. The primary example of this is type-safe programming and types-based approaches to security such as TAL. Program rewriting is a complementary set of techniques which aim to enforce security by rewriting programs to conform to a security policy. Inlining security monitors is an example of this class of techniques. The primary advantage of the language-based approach to security is that it is flexible and can easily express fine-grained security policies.

3.1 Using a Trusted Virtual Machine for Attestation

We propose making remote attestation more flexible and expressive by leveraging language-based techniques and virtual machines. The goal is to *attest program behavior, not a particular binary.* Our domain is platform-independent mobile code that runs on a virtual machine. Instead of a trusted operating system, we use a *trusted virtual machine*, or TrustedVM, to attest to remote parties various properties of code running within it. Software up to, and including, the virtual machine, still has to be trusted, and attested using the standard "signed-hash" method.

In this paper, we use the term virtual machine (VM) to mean a language-based virtual machine that executes some form of platform-independent code. The most widespread example of such a virtual machine is the Java Virtual Machine. It is important to differentiate this from virtual machine monitors (VMM) that virtualize a hardware architecture, and present an interface exactly like, or very similar to, raw hardware. Examples of VMMs are VMWare, and Disco [9].

Requests for remote attestation are now made to the TrustedVM. These requests ask for the enforcement or checking of specific security policies on code that is being run by the TrustedVM. Thus, what is enforced is not the execution of a particular binary, but security policies and other constraints specified by a remote party.

There are two key observations that enable our TrustedVM approach:

- Firstly, code expressed in a portable representation (e.g. Java bytecode) contains high-level information about the code — such as its class hierarchy, method signatures, typing information etc. The presence of this high-level information, as well as the fact that code is expressed in a platform independent format, makes such code very amenable to program analysis. A trusted virtual machine can attest to high-level meta-information about a program, as well as the results of some code analysis done on a program.
- Secondly, code runs completely under the control of a virtual machine, and so its execution can be explicitly monitored. Thus, a trusted virtual machine can attest to dynamic properties regarding the execution of a program, or its inputs.

Some examples of properties that a trusted virtual machine can attest are:

- **TrustedVM attests properties of classes:** the remote party may require class A to subclass a well-known class B, or some interface C. This may be because extending B or C constraints the behavior of A in some way. For example, C may be a restricted interface for input-output operations, that disallows arbitrary network connections.
- **Attesting dynamic properties:** the program being attested runs under complete control of a TrustedVM. Thus, a TrustedVM can

attest to dynamic properties. This includes the runtime state of the program and properties of the input of the program.

- **Attesting arbitrary properties:** A TrustedVM has the ability to run arbitrary analysis code(within the limits set by the security policy of the local host) on the program being attested on behalf of the remote party. Thus the remote party can test for a wide variety of properties by sending across code that does the appropriate analysis.
- **Attesting system properties:** a remote party can send across code that tests certain relevant system and virtual machine properties, and the TrustedVM can attest its results. For example, before running a distributed computing program (such as SETI@Home, or Folding@Home), the server may want to test the floating point behavior of the system and virtual machine by having the TrustedVM run a test suite of floating point programs.

Note that this sort of attestation is a much more fine-grained and semantically richer operation than signing the hash of an executable image — we call this *semantic remote attestation.* What is now attested is not the presence of a particular binary executable, but relevant properties of a program. This has the effect of explicitly separating two concerns that were earlier merged into one — identity and behavior. Claims about code behavior are now made by the trusted virtual machine explicitly checking or deriving them.

A direct consequence of this is that now a variety of different implementations of some functionality will be able to function within our remote attestation framework — as long as they satisfy the properties required of them. Cryptography now plays the part of binding this claim about code behavior to an entity which is qualified to make such claims — a trusted virtual machine.

3.2 Advantages of Semantic Remote Attestation

Semantic remote attestation has a number of advantages over traditional remote attestation:

- It overcomes the most critical shortcoming of traditional remote attestation — semantic attestation reasons about the behavior of the code, without tying that reasoning to a particular executable binary.
- It is dynamic in nature — it can attest to various dynamic properties of a program, such as its runtime state at interesting program points, or input to the program. As opposed to traditional remote attestation, it is not one-time.
- It is flexible — a TrustedVM can carry out arbitrary code analyses and attest its results.

Semantic remote attestation done in virtual machines with platform-independent code enables truly new functionality that was not possible before, because this sort of high-level attestation of program properties cannot be done using native code. Firstly, native code is too low-level, and does not have enough high-level structure and information to drive the sort of analysis our TrustedVM does. Secondly, some of the functionality of a TrustedVM (such as a server sending code to analyze or monitor the program being run) requires platform-independent code.

Semantic attestation can allow for the possibility that some of the properties required of a program may not be satisfied. In that case, the remote party can lower its expectations of how trustworthy the behavior of the program is likely to be, and proceed accordingly, rather than terminate the whole protocol altogether. Thus, properties that make an application trustworthy can now be thought of as *falling within a range, rather than one all-or-nothing attestation.* This has an important consequence for backward compatibility. Most common network protocols today (TCP, HTTP) assume a completely untrusted client. Using a flexible approach to attestation allows these untrusted protocols to be gradually endowed with trusted capabilities, and the ability to deal with clients of varying trustworthiness.

Semantic remote attestation also allows attestation without locking the client into a particular platform, or binary. By far the most scathing critiques of trusted computing have focused on the idea of remote attestation being used to lock consumers into a particular platform, thus extending monopoly control[6]. Semantic remote attestation, however, explicitly separates identity from behavior, and allows flexible attestation of code properties, while

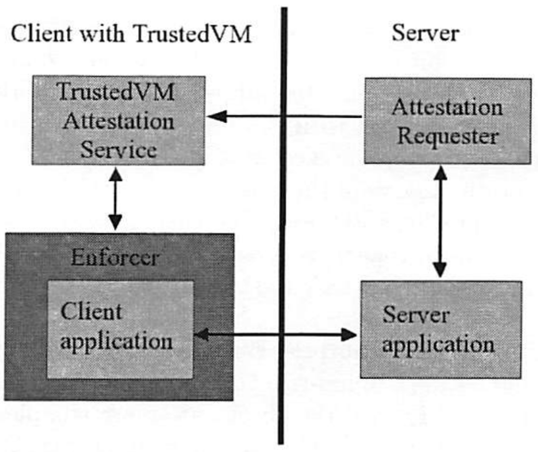


Figure 1: Top-level architecture for dynamic checking in a TrustedVM

allowing inter-operation of a variety of implementations that satisfy these properties.

4 Implementation and Results

Semantic remote attestation is a dynamic process, and attestation of properties can be done at various points during the lifetime of a distributed application. We refer to the machine running the TrustedVM as the client, and the machine which is interested in attesting programs on it as the server. The TrustedVM on a client machine runs an attestation service, whose job is to check or analyze the behavior of applications running under the TrustedVM, and answer attestation requests from other machines. The server has two channels of communication with the client: one with the client application, and another with the attestation service, with which the server communicates to find out about the behavior of the client application. This is shown in Figure 1. The communication between the server and attestation service must be secure and must guarantee integrity and authenticity. This can be done using various cryptographic protocols, such as SSL.

Semantic attestation certifies properties of programs running in a virtual machine. The virtual machine itself, however, runs on top of an operating system. To certify the layers of software up to, and including the virtual machine, still requires traditional remote attestation. This is done in the standard way using signed hashes.

The trusted base of a virtual machine also includes its standard libraries. Even if the virtual machine itself is trustworthy, an adversary could modify the standard system libraries for malicious ends — for example, by substituting the standard security manager class with a weaker one. Thus, the operating system certifies both the virtual machine binary, and the libraries it uses.

At the time of this writing, we did not have access to trusted hardware. Thus, the certificate chains we emit are not rooted in a trusted hardware module, but in the operating system. We do not foresee any conceptual difficulties in interfacing to real trusted hardware, since it is a matter of extending the certificate chain.

We now explain the implementation of two applications that we implemented on our prototype TrustedVM.

4.1 A peer-to-peer protocol

To gain experience with building and using a TrustedVM for semantic attestation, we chose peer-to-peer networks as one example domain. In particular, we considered the Gnutella P2P protocol[2]. Peer-to-peer communication is particularly interesting as a trial application for remote attestation. It is inherently distributed in nature, and current protocols vest a tremendous amount of trust in clients for correct operation of the network. This has led to various security and policy violations, and has even been used to stage denial-of-service attacks [26, 13, 8]. In the case of the Gnutella protocol, the reason for these security weaknesses is that peers believe each other without verification. For example, when given the result of a query(a *query hit* in Gnutella terminology), there is no guarantee that the given machine will actually have the content asked for — the query result can be easily faked and is not checked. This can be exploited to mount a denial-of-service attack against a particular machine by simply flooding the network with query results all pointing to that one machine. The same holds for routing messages that announce which machines are part of the P2P network. These routing messages (called *pong* messages in Gnutella, because they are sent out as replies to ping messages to find out network topology) can also easily be faked.

Our implementation is based on the Java Virtual Machine. We used the Java Development Kit version 1.4.1 from Sun, running on an Intel Pentium-M 1.3 Ghz machine with 384 MB of RAM.

We modified the standard Java network socket class so that network communication over sockets could be captured and monitored. An API is exposed to do this. This captured traffic is sent to a protocol watcher which checks the protocol messages for conformance with a security policy, and informs the server accordingly. The Java bytecode for protocol watcher itself is sent by the server to the attestation service on the client, since it embodies the policy the server is interested in enforcing on the client.

Here we see the utility of using a machine-independent code representation — the protocol watcher can be sent to and executed on various platforms, as long as they have a Java TrustedVM. Note that the protocol watcher is independent of the application. Various implementations of a protocol can be checked by same protocol manager.

We implemented a protocol very similar to Gnutella, though somewhat simpler in its wire format (to simplify debugging and parsing of network messages). The protocol watcher examined all outgoing messages from the client, and checked two properties:

- for *pong* messages, make sure that the IP address in the message is the same as that of the client.
- for *query hit* messages, make sure that the file that is mentioned in the query actually exists on the client.

These two properties were checked in particular because they can exploited to mount denial-of-service attacks, and can be easily spoofed — they are completely unchecked.

We have not yet implemented a protocol checker for multi-hop messages, but this can be done using transitive certificate chains.

At the time of this writing, we are not aware of any applications that use functionality provided by trusted hardware, so we do not have a meaningful comparison benchmark. However, we can measure the overhead that protocol checkers impose on network latency. To measure this, we measured the time it took for 5000 subsequent ping/pong, and query/queryhit messages to travel between two machines. Both were on a 100 Mbps Ethernet network. This was compared with the same benchmark run without a protocol checker. Within experimental error, the timings were the same — between 9.5 and 10.2 seconds for both cases. Increasing the number of iterations to 10,000 yielded similar results — both cases took between 18.5 and 20.8 seconds.

This result is not a surprise since the checks are simple, and network round-trip-time dominates computation time. Even if the checks are more complex, in realistic situations they are only done periodically, and not in a tight loop. Thus, end-to-end performance of applications is still likely to show very little overhead.

We can distinguish between two kinds of applications that run on a TrustedVM. Legacy applications are those that do not explicitly make use of trusted functionality such as remote attestation. We just presented an example of a legacy untrusted application being attested by a TrustedVM, completely transparently. New applications that are written with trusted functionality in mind can use a much broader range of a TrustedVM's facilities.

We also unsuccessfully tried writing protocol watchers by using bytecode rewriting[12]. This turned out to be unsuitable for the task because load-time rewriting of system classes is not allowed in the JVM. Rewriting bytecode at load-time requires a specialized class loader, that transforms the bytecode before handing it off to the virtual machine. However, all system classes (`java.lang.*`, `java.net.*` etc) must be loaded by the "primitive" system classloader. Systems such as SASI[14] and Naccio[15] inline reference monitors by rewriting Java bytecode. However, they transform the classfiles off-line, and not at load-time.

4.2 A distributed computing application

The previous application was an example of dynamic enforcement of security properties within a TrustedVM. A TrustedVM can also attest static properties, of both the system it is running on, as well as the code that it runs. The attestation requester sends across code for testing various properties, which the TrustedVM then executes. The results are signed, and sent back to the attestation requester (see Figure 2). The results of these tests can then affect further computation and communication between the two parties.

This is useful for distributed computing. There are a number of popular distributed computing projects, such as SETI@Home [20] and Folding@Home[27], that distribute work units out to a large number of clients. They face a common problem — getting some assurances about the behavior and capabilities of their numerous clients. There is a complex trust relationship between the server and client, since the server expects the client to use a particular algorithm, compute answers within certain bounds, and not return maliciously crafted or incorrect answers. Currently, this problem is solved to some extent by measures such as redundancy and keeping track of client behavior over a period of time[22].

Using a semantic remote attestation framework can benefit a distributed computing application in the following ways:

- The server can test various properties of both the system, as well as the client, by having the TrustedVM execute tests for it. This would give the server a better idea of the capabilities of the platform as well as the client. This knowledge can be used both for giving the client suitable work units, and estimating how "good" its answers are likely to be.

- Testing for properties in this way makes the trust relationships between the client and server explicit. Now, instead of being implicit and being taken on trust, they are explicitly enforced or checked by the TrustedVM.

To experiment with these concepts, we took an existing distributed computing project, examined its client-server trust relationships, and re-implemented it to run on our TrustedVM. We chose a distributed computing project that tries to find large Mersenne Primes — the Great Internet Mersenne Prime Project, or GIMPS[4]. Mersenne primes are prime numbers of the form $2^n - 1$. Just like SETI@Home[20], GIMPS distributes its computation over a large number of clients on the Internet. We chose GIMPS because it divides the problem into three subproblems, each with different computational needs. They are:

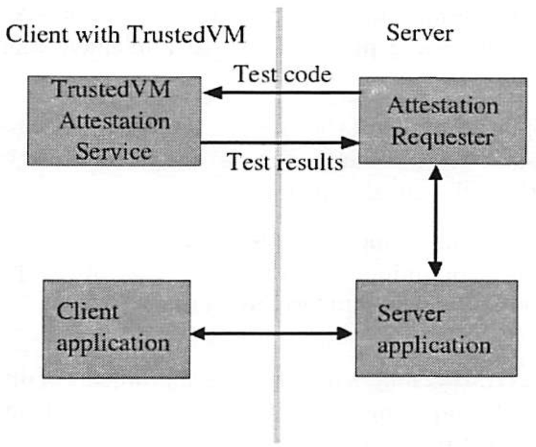


Figure 2: Top-level architecture for using test suites in a TrustedVM

1. First time primality check: This is the most computationally intensive of the three problems and is assigned to the fastest clients. It also requires double precision floating point support for doing a fast Fourier transform (FFT).

2. Double check assignment: It verifies the results of the first time primality check. The workload is smaller in this case and hence is assigned to slower clients. It also requires double precision floating point support.

3. Factoring work: This tries to eliminate a few test candidates by finding small factors using some common factoring algorithms. This reduces candidates for more expensive primality checks. Since this least expensive of the three this gets assigned to the slowest clients.

For a full treatment of the mathematical background of this problem see [4].

Many different implementations of the client exist [1]. The various clients differ from each other in the following ways

- The clients have subtle differences in the algorithms used.

- Some of the clients are highly specialized for a particular architecture and hence lose portability. Others have to be compiled for various architecture/OS combinations.

- The performance of different clients differ, even on the same platform due to implementation differences.
- The accuracy of the clients also differ hence results slightly vary. This is mainly because of the different algorithms used.
- More surprisingly the results differ for the same client on different platforms because of the differences in the underlying hardware.

Thus, GIMPS suffers from the same problems of distributed computing as pointed out above: its client-server trust relationships are implicit, and the server has very little information about the capabilities of the clients. A client is expected to behave reasonably because it is specific to a particular platform, and its behavior has been tested on that platform.

These problems can be solved by running this application on a TrustedVM. The server now explicitly tests the relevant capabilities of the client-side by asking the TrustedVM to execute a *test suite*, and return the attested results. This solution also has the added advantage of being portable across any range of architectures that implement a TrustedVM.

The two capabilities that are relevant for the GIMPS project are: floating point precision and the computational power. Our test suite has a compoenent for each.

The most computationally intensive routine in the prime factorization problem is computing a fast Fourier transform and its inverse. Hence to test for computation speed we execute a one-dimensional fast Fourier transform over small but typical data and time it. The result of this test helps the server to give the client an appropriate work unit.

For floating point precision we are interested in testing if the platform implements a double precision floating point operations and also if it complies with the IEEE 754 standard for floating point. In particular we used the Java port of the Elefunt test suite[3]. Elefunt[11] is a test suite to check for the compliance of floating point implementation of the various functions with the IEEE floating point standards. Since the FFT and the inverse FFT use the sine and the cosine functions the two tests that we are interested in are the ones that determine the accuracy of these functions. Depending on results we determine whether the client is accurate enough to run the computations.

Thus, the server now has reliable information about the clients it is communicating with. Using this information, it can both vary the work units given out to clients, as well as make estimates about the accuracy of their answers. Moreover, the whole application is now portable, and does not depend on specific clients.

5 Related Work

To the best of our knowledge, there is no prior work that aims to make the mechanism of remote attestation more fine-grained, dynamic and platform-independent. However, the field of trusted computing has attracted a great deal of attention recently.

Garfinkel et. al.[17] have proposed the TerraVM[16] virtual machine monitor architecture to interface with underlying trusted hardware. Their architecture provides two VMM abstractions to software — an open box VMM, and a closed box VMM. The open box VMM simply provides a legacy, untrusted interface. This allows old operating systems and software to run unmodified on it. The closed box VMM, however, provides an interface to underlying trusted hardware that new software can use. A number of such VMMs can execute on bare hardware. They are strongly isolated from each other, and have their own encrypted storage.

There are many important differences between TerraVM and our TrustedVM architecture. While TerraVM provides an interface just like real hardware, the TrustedVM exposes a much higher-level interface. It executes platform-independent code. Their goal is providing strong isolation between applications running in different VMMs, ours is to provide a better technique for remote attestation. Most importantly, TerraVM uses the standard "signed-hash" method of remote attestation. The authors acknowledge some of the shortcomings of standard remote attestation, and call for "appropriate technical and legal protections ... to minimize abuse".

The goal of TerraVM is similar to Microsoft's Palladium architecture. Palladium is said to have a high-assurance trusted microkernel running on hardware (called the *nexus*) that provides strong isolation between legacy untrusted and newer trusted applications, as well as among trusted applications. Unfortunately, to our knowledge there is no published technical documentation about Palladium, which makes it hard to make an in-depth comparison.

The Cerium system[10] aims to provide tamper-evident execution. The goal is to know if the remote execution of a program was carried out in an untampered manner. They use a physically tamper-resistant CPU block. This executes all code — it is not a co-processor. Tamper-resistance is provided both at the hardware level, by physical means, and at the software-level, by encrypting all data (including cache data that is written back to main memory). Semantic remote attestation is orthogonal to this — a TrustedVM could use Cerium as an underlying hardware architecture.

The Digital Distributed System Security Architecture[18] had many of the features of today's TCPA specification[5], including secure bootstrapping, and remote attestation of system software using signed hashes. The Aegis system[7] provided secure bootstrapping. Every layer of software from the hardware up was checked (using stored hashes) and control was passed to it only if it was untampered. Both these systems did not focus on improving remote attestation.

The Trusted Computing Group (TCG), has begun producing specifications of a hardware trusted module to be used in personal computers[5]. They call it a trusted platform module (TPM). Some models of IBM ThinkPad laptops contain a similar module. A commodity trusted computer will couple this trusted module with software that provides secure booting and strong isolation.

6 Discussion and Future Work

There are many avenues for further investigation. Protocol watchers and test suites, as presented here, are only two of many kinds of expressive attestation a TrustedVM is capable of.

A TrustedVM is capable of attesting the results of some static analysis done on code. However, there are not many static analyses of code for properties of interest to a remote server. Most static analyses and runtime enforcement policies so far have been geared towards protecting a host from malicious mobile code. Thus, the emphasis has been on type-safety, information-flow, and resource control and other safety issues. The emphasis is different for remote attestation. Servers want to know if the application is obeying some high-level semantic rules. One candidate for an analysis that may be of interest to servers is information flow[23]. Such an analysis would convince the server that a client is not leaking the results of some confidential computation, or data.

In our current implementation, the policy a server wants enforced is embodied in the protocol watcher or test suite. We would like to develop a systematic language for expressing remote attestation requests. With "signed-hash" attestation, this was not an issue. But a TrustedVM provides a broad range of fine-grained attestation capabilities, and a language is probably the right tool to make full use of them.

We would also like to gain experience with developing more applications that use the functionality provided by a TrustedVM. Distributed firewalls[19] implement a network traffic policy at the end-points of a network, rather than at one single point. This way of distributing trust seems like a good match for implementation on a TrustedVM.

The ability to communicate to a server what particular property of a program could not be certified can be very useful. Using TrustedVMs, this information can be communicated, and the server can get detailed information about what desired properties are not present in a client program. It can then make an informed decision about either decreasing its trust in this particular instantiation of a protocol, or stopping altogether. Thus, the server gains some dynamic feedback about the trustworthiness of its clients. We believe this property can be fruitfully exploited to "port" a variety of untrusted network protocols (TCP, HTTP etc) to a trusted computing framework in a gradual manner, and yet have various implementation of them inter-operate. This is in stark contrast to the all-or-nothing model that standard "signed-hash" remote attestation provides — attestation either passes or fails — there is no gradation. This would also provide a gentler upgrade path for applications as trusted hardware becomes increasingly available in the market.

7 Conclusion

Standard ways of doing remote attestation are based purely on cryptography, and suffer from many critical shortcomings — they are static, inexpressive, inflexible and do not scale. Most importantly, they do not speak about program behavior — they can only attest to the presence of a particular binary. It is possible for an attested binary to have bugs and not obey the security policy a server was expecting it to. Remote attestation is hard to scale up to a flood of software patches and upgrades. It also does not accommodate a varied, homogeneous computing environment very well.

We have introduced a novel technique for remote attestation based on language-based virtual machines. The core idea behind our technique, called semantic remote attestation, is to use a language-based virtual machine that executes a form of platform-independent code. Software up to and including this virtual machine is trusted. However, the virtual machine can certify various properties of code running under it by explicitly deriving or enforcing them. This can be done in many ways, such as observing the execution of programs running in a VM, or analyzing the code before execution. This is particularly easy to do with high-level platform-independent code that has a lot of information about the structure and properties of code.

The fact that "trusted computing", and its core technique, standard remote attestation, can lock consumers into a particular program or platform has been a very widely expressed fear. A key advantage of our approach is that reasoning about the behavior of a program is now not tied to a particular binary. Semantic remote attestation checks for program properties, and works with different implementation of the same program as long as they satisfy the properties required of them.

To validate our ideas we have implemented a prototype TrustedVM by modifying a Java virtual machine to observe the behavior of network protocols. We have used this prototype to check the untrusted behavior of a Gnutella-like peer-to-peer protocol. Specifically, we check that messages about network topology and query results are not spoofed. Our measurements show that the overheads of checking are negligible, at least for the simple checking this particular application needs. However, even the few simple checks we do are sufficient to overcome some of the most well-known weaknesses in peer-to-peer protocols.

Trusted computing introduced the concept of remotely supervised execution - the idea that a remote server will be able to monitor as well as change the execution of a program on a client machine. Remote attestation is the key to doing this. However, current architectures for attestation are able to implement this idea in only a very limited way. Semantic remote attestation takes this idea to its logical conclusion — to have fine-grained, dynamic control over the monitoring and execution of an application.

References

[1] GIMPS source code timings page. `http://hogranch.com/mayer/gimps_timings.html`.

[2] The gnutella protocol specification. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.

[3] Java port of elefunt. http://www.math.utah.edu/ beebe/software/java/

[4] The Great Internet Prime Mersenne Search: GIMPS. http://www.mersenne.org/.

[5] T. C. P. Alliance. TCPA PC-specific implementation specification (http://www.trustedcomputing.org), May 2001.

[6] R. Anderson. Cryptography and competition policy — issues with trusted computing. In *Workshop on Economics and Information Security*, May 2003.

[7] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.

[8] S. Bellovin. Security aspects of napster and gnutella. In *USENIX Security Symposium*, Aug. 2000.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[10] B. Chen and R. Morris. Certifying program execution with secure processors. In *USENIX HotOS Workshop*, May 2003.

[11] W. Cody and W. Waite. *Software manual for the elementary functions.* Prentice Hall, 1980.

[12] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[13] Z.-Y. Demetris. Exploiting the security weaknesses of the gnutella protocol, Mar. 2002.

[14] Erlingsson and Schneider. SASI enforcement of security policies: A retrospective. In *NSPW: New Security Paradigms Workshop.* ACM Press, 2000.

[15] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[17] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible os support and applications for trusted computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2003.

[18] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[19] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[20] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. Seti@home — massively distributed computing for SETI. *Computing Science and Engineering*, 3(1), 2001.

[21] D. Kozen. Language-based security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.

[22] D. Molnar. The SETI@home problem. http://www.acm.org/crossroads/columns/onpatrol/september2000.html, Sept. 2000.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[24] G. C. Necula. A scalable architecture for proof-carrying code. *Lecture Notes in Computer Science*, 2024:21+, 2001.

[25] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Lecture Notes in Computer Science*, 2000:86–??, 2001.

[26] D. Wallach. A survey of peer-to-peer security issues. In *International Symposium on Software Security*, Nov. 2002.

[27] B. Zagrovic, E. Sorin, and V. Pande. Beta hairpin folding simulations in atomistic detail using an implicit solvent model. *Journal of Molecular Biology*, 317(4), 2002.

Towards Scalable Multiprocessor Virtual Machines

Volkmar Uhlig Joshua LeVasseur Espen Skoglund Uwe Dannowski
System Architecture Group
Universität Karlsruhe
contact@l4ka.org

Abstract

A multiprocessor virtual machine benefits its guest operating system in supporting scalable job throughput and request latency—useful properties in server consolidation where servers require several of the system processors for steady state or to handle load bursts.

Typical operating systems, optimized for multiprocessor systems in their use of spin-locks for critical sections, can defeat flexible virtual machine scheduling due to lock-holder preemption and misbalanced load. The virtual machine must assist the guest operating system to avoid lock-holder preemption and to schedule jobs with knowledge of asymmetric processor allocation. We want to support a virtual machine environment with flexible scheduling policies, while maximizing guest performance.

This paper presents solutions to avoid lock-holder preemption for both fully virtualized and paravirtualized environments. Experiments show that we can nearly eliminate the effects of lock-holder preemption. Furthermore, the paper presents a scheduler feedback mechanism that despite the presence of asymmetric processor allocation achieves optimal and fair load balancing in the guest operating system.

1 Introduction

A recent trend in server consolidation has been to provide virtual machines that can be safely multiplexed on a single physical machine [3, 7, 24]. Coupling a virtual machine environment with a multiprocessor system furthers the trend of untrusted server consolidation.

A multiprocessor system offers many advantages for a virtualized environment. The hypervisor, the controlling agent of the virtual machine environment, can distribute the physical processors to guest operating systems (OS) to support arbitrary policies, and reassign the processors in response to varying load conditions. The allocation policy may support concurrent execution of guests, such that they only ever access a fraction of the physical processors, or alternatively time-multiplex guests across a set of physical processors to, e.g., accommodate for spikes in guest OS workloads. It can also map guest operating systems to virtual processors (which can exceed the number of physical processors), and migrate between physical processors without notifying the guest operating systems. This allows for, e.g., migration to other machine configurations or hot-swapping of CPUs without adequate support from the guest operating system. It is important to recognize that allowing arbitrary allocation policies offers much more flexibility than schemes where one can only configure a virtual machine to either have an arbitrary share of a single processor [7, 24], or have uniform shares over multiple physical processors [10, 24].

Isolating commodity operating systems within virtual machines can defeat the assumptions of the guest operating system. Where the guest operating system expects constant resource configurations, critical timing behavior, and unrestrained access to the platform, the virtual machine provides illusionary access as it sees fit. Several methods exist to attempt to satisfy (a subset of) the assumptions of the guest operating system. The solutions may focus on the issues of instruction set emulation, such as trapping on system instructions [22], or they may focus on the behavior of the guest operating system algorithms, such as dynamic allocation of physical memory [25].

This paper presents solutions to two problems that arise with scheduling of virtual machines which provide a multi-processor environment for guest operating systems. Both problems limit scalability and performance. First, guest operating systems often use spin-locks as a means to offer exclusive access to code or data. Such spin-locks are, by design, only held for a short period of time, but if a virtual machine is preempted while holding the lock this assumption no longer holds. The crux of the problem is that the same virtual machine may still be running on a different processor, waiting for the lock to be released, thus wasting huge amounts of processor cycles (often several milliseconds).

The second problem is due to the ability of virtual processors to offer the illusion of varying speed. Today's operating systems cannot react well to multi-processor systems where otherwise identical CPUs have asymmetric and varying clock speeds. The problem manifests itself in suboptimal scheduling of CPU intensive workloads and burst-load handling.

To address the first problem, we have devised two techniques for avoiding preemption of lock holders, one requiring modifications of the locking primitives in the guest OS, and one in which the hypervisor attempts to determine when it is safe to preempt the virtual machine (i.e., without modifying the guest OS). Initial results suggest that our lock holder preemption avoidance schemes can increase high-load web server performance by up to 28% compared to an approach where the virtualization layer does not take guest OS spin-locks into account.

To handle asymmetric CPU speeds we propose a technique called *time ballooning* where the hypervisor coerces the guest OS to adapt scheduling metrics to processor speed. The coercion may manifest as the introduction of ghost processes into the scheduling queues of the guest OS, or as balloon processes which donate their cycles back to the virtualization layer when scheduled by the guest OS. By artificially increasing the load on a virtual CPU we pressure the guest OS into migrating processes to other virtual CPUs with more available resources.

The remainder of the paper is structured as follows. Section 2 elaborates on the problem of lock-holder preemption. Sections 3 and 4 describe our solutions with lock-holder preemption avoidance and time ballooning, followed by experimental results in Section 5. The implications of our solution and future work are discussed in Section 6. Section 7 presents related work, and finally Section 8 concludes.

2 The Case for Lock-holder Preemption Avoidance

Many of the commodity operating systems used in server consolidation have optimized support for multiple processors. A primary function of the multiprocessor support is to guarantee atomic and consistent state changes within the kernel's data structures. Typical kernels use memory barriers to ensure in-order memory updates, and they craft critical sections protected by locks to enforce atomic updates. The critical sections may be associated with a region of code, or as with more fine grained locking, they may be associated with a particular piece of data.

When the number of processors in a system increases, more processors will be competing for access to the critical sections. To achieve multiprocessor scalability it is important that the time a processor spends in a critical section is short. Otherwise, the processors trying to acquire the lock for the critical section can experience long waiting times. Designing a system for short lock-holding times makes it feasible to poll for a lock to be released (i.e., using spin-locks). Short lock-holding times may also obviate the need to implement more expensive locking primitives to enforce fair lock access, since the kernel may achieve such fairness statistically.

A very different approach to achieve multi-processor scalability in operating systems has been to avoid locking altogether by using non-blocking synchronization primitives. Although an operating system kernel can in theory be made lock free using atomic *compare-and-swap* instructions supported by many hardware architectures, it has been shown that special hardware support is needed to make lock free kernels feasible [11]. Such special hardware support has been used to implement lock-free versions of Synthesis [19] and the Cache-kernel [6], but is not applicable to commodity operating systems in general, both because of the hardware requirements and the tremendous task of rewriting large parts of the kernel internal data structures and algorithms. Some form of locking therefore seems unavoidable.

When running a commodity operating system in a virtual machine, the virtual machine environment may violate some of the premises underlying the guest operating system's spin-locks. The virtual machine can preempt the guest kernel, and thus preempt a lock holder, which can result in an extension of the lock holding time. For example, in Linux, the typical lock holding time is under 20 microseconds (see Figure 2), which a preemption can easily extend by several time slices, often in the order of tens of milliseconds.

Consequently, the main effect of lock preemption is the potential for wasting a guest operating system's time slice. If a guest kernel spins on a preempted lock, it could live out the remainder of its time slice spinning and accomplishing no work. Thus spin-times are amplified. A concomitant effect is the violation of the original statistical fairness properties of the lock.

The side effects of lock holder preemption could be avoided with coscheduling [21]. In coscheduling (or gang scheduling), all virtual processors of a virtual machine are simultaneously scheduled on physical processors, for an equal time slice. The virtual machine could preempt lock holders without side effects, since the coschedule guarantees that another processor will not spin on a preempted lock. However, coscheduling introduces several problems for scalability and flexibil-

	Povray	Kbuild	Apache 2
total	0.04%	15.3%	39.2%
average	3.0μs	1.8μs	2.2μs
max	103μs	293μs	473μs
std.dev.	5.3μs	6.7μs	7.7μs

Table 1. Lock-holding times for various Linux workloads. Hold times are measured while at least one kernel lock is being held by the CPU.

ity. Coscheduling activates virtual processors whether or not they will accomplish useful work, easily leading to underutilized physical processors. Further, coscheduling precludes the use of other scheduling algorithms, such as multiplexing multiple virtual processors on the same physical processor (e.g., in response to fault recovery of a failed processor, or load balancing).

Alternative lock wait techniques to spinning, such as reschedule or spin-then-reschedule, have successfully been applied to user applications [16], but these techniques are generally not applicable to traditional operating systems code because the kernel does not always enjoy the option of preempting its current job (e.g., if within a low-level interrupt handler). To conclude, we see that spin-locks are and will be used in commodity operating systems, and preempting lock-holders *may* as such pose a significant performance problem.

To determine whether the frequency of lock-holder preemption really merits consideration, we instrumented Linux 2.4.20 with a lock tracing facility to inspect locking statistics. Table 1 shows the results for three workloads with varying characteristics from the application spectrum. We measured the time for when a CPU holds at least one lock, on a machine with four 700MHz Intel Xeon processors. With CPU-intensive applications (povray, a ray-tracing application) we found locks being held for an average of $3.0\mu s$ and a maximum of $103\mu s$. With an I/O-intensive workload like the Apache 2 web server under stress these numbers were $2.2\mu s$ and $473\mu s$ respectively.

From our tracing experiments we observe that the probability of preempting a virtual CPU while holding a lock lies between 0.04% for CPU-intensive workloads and 39% for I/O-intensive workloads. These numbers indicate that scheduling a virtual machine running an I/O-intensive workload without regard for guest OS spin-locks can severely impact the performance of the virtual machine. Some scheme for dealing with lock-holder preemption is therefore deemed necessary.

There are two approaches to deal with lock-holder preemption. The first approach is to detect contention on a lock and to donate the wasted spinning time to the lock holder. Also known as *helping locks*, this approach requires substantial infrastructure to donate CPU time between virtual CPUs (provided donation is possible at all) [14]. The second approach is to avoid preempting lock-holders altogether. Instead, soon to become lock-holders are preempted before acquiring a lock, or preemption is delayed until after the last lock has been released.

Depending on the level of virtual machine awareness in the guest operating systems, different methods of lock-holder preemption avoidance can be used. We discuss these methods in the following section.

3 Lock-holder Preemption Avoidance

Lock holder preemption avoidance can be achieved by either modifying the guest operating system to give hints to the virtual machine layer (intrusive), or have the virtual machine layer detect when the guest operating system is not holding a lock (non-intrusive). The former approach is well suited for systems where the virtualized architecture is not identical with the underlying hardware; also called *paravirtualization* [28]. The latter approach is well suited for fully virtualized systems.

3.1 Intrusive Lock-holder Preemption Avoidance

For the intrusive approach we use a similar scheme as implemented in Symunix II [8] and described by Kontothanassis et. al. [17]; the main difference being that the guest OS itself is here the application giving scheduling hints to the lower layer (the hypervisor).

Intrusive lock-holder preemption avoidance in our system is achieved by augmenting Linux (our guest OS) with a delayed preemption mechanism. Before acquiring a lock, the guest OS indicates that it should not be preempted for the next n microseconds. After releasing the lock, the guest OS indicates its willingness to be preempted again. The virtualization layer will not preempt the lock-holder if it has indicated that it wishes no preemption, but rather set a flag and delay the preemption by n microseconds. When the guest OS releases the lock, it is required to check the flag, and if set, immediately yield its processing time back to the virtualization layer. Failure to give back the processing time will be caught by the virtualization layer after n microseconds, and the guest operating system will be penalized by a subsequent reduction in its processing time and the possibility of untimely preemption.

The value to choose for n depends on how long the guest OS expects to be holding a lock and will as such rely heavily upon the operating system used and the

	Povray	Kbuild	Apache 2
Locked:			
total	0.04%	15.3%	39.2%
average	3.0μs	1.8μs	2.2μs
max	103μs	293μs	473μs
std.dev.	5.3μs	6.7μs	7.7μs
Unsafe:			
total	0.09%	26.6%	98.9%
average	6.9μs	17.8μs	1.4ms
max	1.4ms	2.0ms	47.6ms
std.dev.	28.7μs	52.4μs	7.5ms

Table 2. Lock-hold and unsafe times

workload being run. We have run a number of lock-intensive workloads on a version 2.4.20 Linux kernel and found that more than 98% of the times the Linux kernel holds one or more locks, the locks are held for less than 20μs. These numbers suggest that setting n any higher than 20μs will not substantially decrease the probability of preempting lock holders in the Linux kernel.

3.2 Non-intrusive Lock-holder Preemption Avoidance

It is not always the case that one has the possibility of modifying the guest operating system, in particular if the kernel is only distributed in binary form. We therefore need non-intrusive means to detect and prevent lock-holders from being preempted. Utilizing the fact that the operating system will release all kernel locks before returning to user-level, the virtualization layer can monitor all switches between user-level and kernel-level,[1] and determine whether it is safe to preempt the virtual machine without preempting lock-holders. This gives us a first definition of *safe* and *unsafe* preemption states:

safe state — Virtual machine is currently executing at user-level. No kernel locks will be held.

unsafe state — Virtual machine is currently executing at kernel-level. Kernel locks *may* be held.

The safe state can be further refined by monitoring for when the guest OS executes the equivalent of the IA-32 HLT instruction to enter a low-latency power saving mode (while in the idle loop). Since the operating system can be assumed to hold no global kernel locks while suspended, it is safe to treat the HLT instruction

[1]The mechanism used for monitoring depends heavily on the hardware architecture emulated and the virtualization approach used. One can for instance catch privileged instructions, or insert monitoring code where the hypervisor adds or removes protection for the guest OS kernel memory.

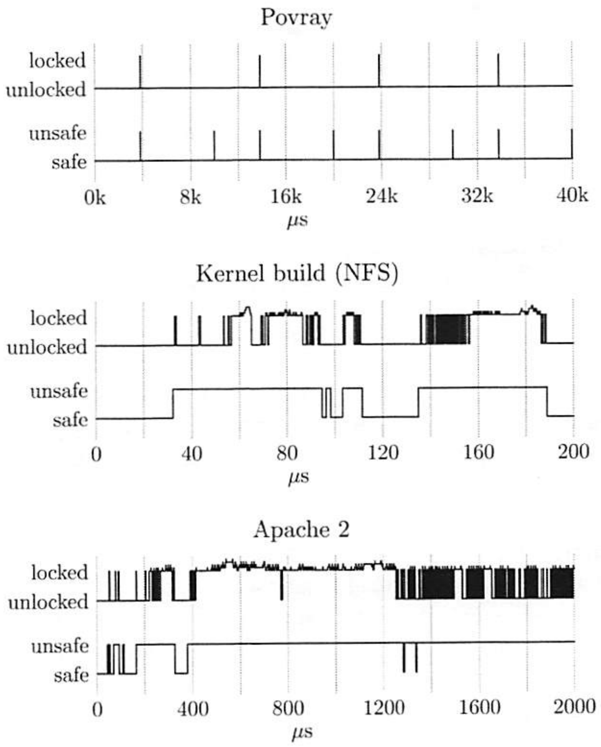


Figure 1. Locked and unsafe times for three different locking scenarios. Povray spends most of the time executing at user-level. Linux kernel build over NFS spends a considerable amount of time at user-level, and moderately stresses the VFS and network layer of the kernel. Apache 2 utilizes the sendfile system call which offloads large amounts of work to the kernel itself.

as a switch to safe state. A switch back into unsafe state will occur next time the virtual CPU is rescheduled (e.g., due to an interrupt).

With the safe/unsafe scheme it is still possible that the virtual machine will be preempted while a *user-level* application is holding a spin-lock. We ignore this fact, however, because user-level applications using spin-locks or spin-based synchronization barriers are generally aware of the hardware they are running on, and must use some form of coscheduling to achieve proper performance. Section 6.4 deals with such workloads in more detail.

In order to substantiate the accuracy of the safe/unsafe model for approximating lock-holding times, we augmented the lock tracing facility described in Section 2 with events for entering and leaving safe states. Our measurements are summarized in Table 2. Figure 1 shows more detailed excerpts of the traces for three different workload classes (spikes on top of the locking states indicate nested locks). The povray work-

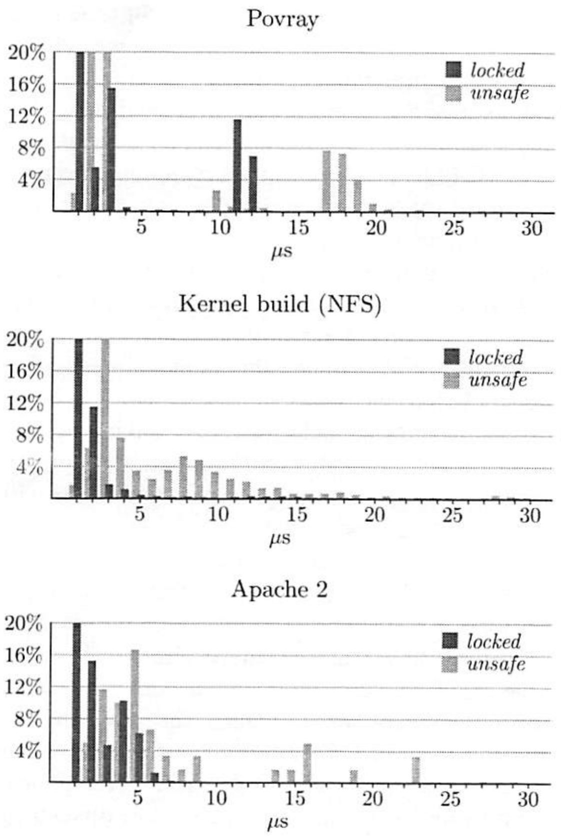


Figure 2. Lock-hold and unsafe state time distributions for three different locking scenarios. Histograms are for readability reasons truncated to a probability of 20%.

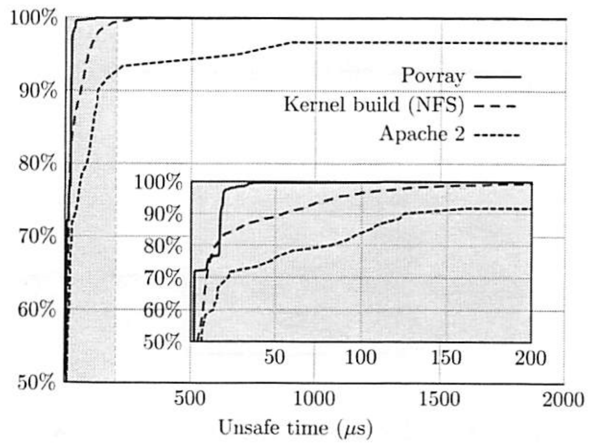


Figure 3. Cumulative probability of unsafe state times for three different workloads. The two graphs show the probabilities of both long and short unsafe state times.

load executes almost entirely at user-level and experiences short unsafe periods only at fixed periods. The kernel-build workload performs a parallel build of the Linux kernel on an NFS-mounted file system. It moderately stresses the VFS subsystem and network layer while spending a fair amount of time at user-level. The figure shows a typical 200μs sample period. Lastly, the Apache 2 workload continually serves files to clients and only sporadically enters safe states (only 1.1% of the execution time is spent in a safe state). The large amount of unsafe time can be attributed to Apache's use of the sendfile system call to offload all file-to-network transfer into the kernel (i.e., avoid copying into user-level).

For the Povray and kernel-build workloads we observe that the unsafe times reasonably approximate their lock holding times. Figure 2 shows that the unsafe times for these workloads are generally less than 10μs longer than the lock-holding times. For the Apache 2 workload, however, the unsafe times are on average three orders of magnitude longer than the locking times. We observe an average unsafe time of more than 1.4ms. This is in spite of locking times staying around 2μs, and can, as mentioned above, be attributed to the Apache server offloading file-transfer work to the kernel by using the sendfile system call.

Now, observing that the standard deviation for the unsafe times in the Apache 2 workload is rather high (7.5ms), we might be tempted to attribute the high disparity between average lock-holding times and unsafe times to a number of off-laying measurements. Looking at Figure 3, however, we see that the Apache 2 workload has substantially longer unsafe times even for the lower end of the axis. For example, the Apache 2 workload only has about 91% of the unsafe times below 200μs, while the povray workload has close to 100% of its unsafe times below 20μs. These numbers suggest that the unsafe state approximation to lock-holding times is not good enough for workloads like Apache 2. We want a better approximation.

Having some knowledge of the guest OS internals, it is often possible to construct points in time, so called *safe-points*, when the virtual machine's guest OS is guaranteed to hold no spin-locks.

One example of how safe-point injection can be achieved is through targeted device drivers installed in the guest OS, designed to execute in a lock free context. An example (compatible with Linux) is the use of a network protocol handler, added via a device driver. The virtualization layer could craft packets for the special protocol, hand them to the virtual NIC of Linux, from where they would propagate to the special protocol handler. When the guest OS invokes the protocol handler, it will hold no locks, and so the protocol handler is safe to yield the time slice if a preemption is pending.

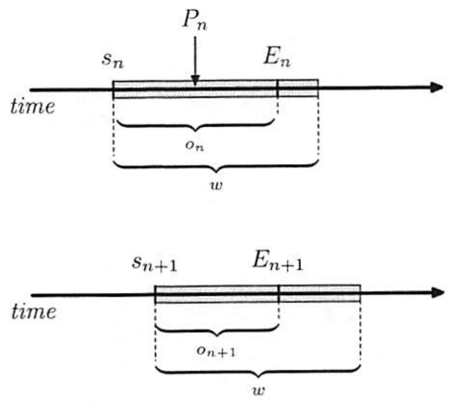


Figure 4. Virtual machine scheduling. A window of size w indicates when a VM may be preempted. E_n indicates the n^{th} end of time slice, P_n indicates the n^{th} actual preemption point, and s_n indicates the start of the n^{th} preemption window.

In an experimental Linux setup we measured as little as 8000 cycles for such a packet to travel from the virtual NIC to our protocol driver under high load. With time slice lengths in the millisecond range, this enables very precise injection of preemption points.

3.3 Locking-Aware Virtual Machine Scheduling

In the following section we describe a mechanism for efficient scheduling of multiple multi-processor virtual machines, leveraging the techniques described in the previous two sections.

Virtual CPUs can be modeled as threads in the virtualization layer which are then subject to scheduling, each one in turn receiving its time slice to execute. Our goal is to preempt a virtual machine (i.e., a thread in the virtualization layer) as close to the end of its time slice as possible, or before that if the virtual machine decides not to consume the whole time slice. In addition, we want to guarantee fairness so that a virtual machine will aggregately obtain its fair share of the CPU time.

Figure 4 illustrates the basis of our virtual machine scheduling algorithm. We define a preemption window, w, around the end of the virtual machine's time slice. For the n^{th} time slice, this preemption window starts at time s_n, E_n is the actual end of the allocated time slice, and P_n is the time when the hypervisor finds it safe to preempt the virtual machine (the last spin-lock was released or the VM entered a safe state). If no safe preemption point occurs before the end of the window, the hypervisor will enforce a preemption.

Our goal with the scheduling algorithm is to choose the window start, s_n, so that the preemption point, P_n, on average coincides with the actual end of time slice, E_n (i.e., $\sum_{i=0}^{n} E_i - P_i = 0$). In doing so we achieve fair access to desired processor time.

Now, assume that in the past the average distance between our start points and preemption points equaled an offset, o_n. In order to keep this property for our next time slice we must calculate the next offset, o_{n+1}, so that it takes into account the current distance between preemption point and window start ($P_n - s_n$). This is a simple calculation to perform, and the result is used to determine the next window start point: $s_{n+1} = E_{n+1} - o_{n+1}$. The consequence of our algorithm is that a preemption that occurs before the end of time slice will cause the preemption window to slide forwards, making premature preemptions less likely (see lower part of Figure 4). Conversely, a preemption after end of time slice will cause the preemption window to slide backwards, making premature preemptions more likely.

The scheduling algorithm ensures that any preemptions before or after the end of time slice will eventually be evened out so that we achieve fairness. However, since the algorithm keeps an infinite history of previous preemptions it will be slow to adapt to changes in the virtual machine workload. To mitigate this problem we can choose to only keep a history of the last k preemptions. The formula for calculating the next window offset then becomes:

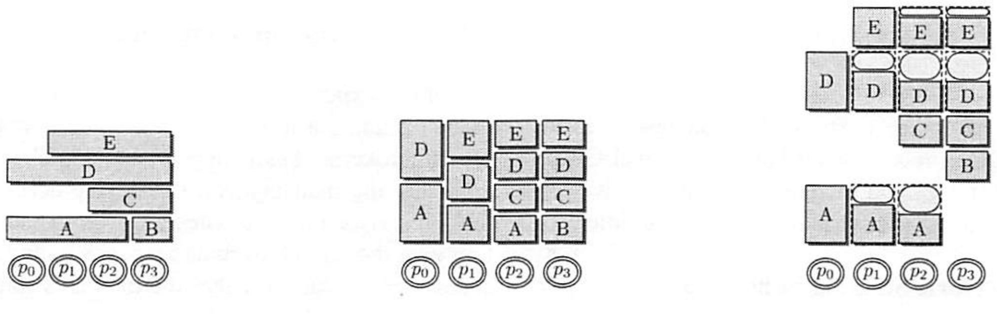
$$o_{n+1} = \frac{o_n(k-1) + (P_n - s_n)}{k}$$

A further improvement of the algorithm is to detect when preemptions have to be forced at the end of the preemption window—a result of no safe state encounters—and remedy the situation by injecting safe points into subsequent preemption windows.

There are two tunable variables in our VM scheduling algorithm. Changing the window length, w, will decrease or increase the accepted variance in time slice lengths, at the expense of having the virtual machines being more or less susceptible to lock-holder preemptions. Changing the history length, k, will dictate how quickly the hypervisor adapts to changes in a virtual machine's workload.

4 Time Ballooning

The scheduling algorithm of a multiprocessor OS distributes load to optimize for some system wide performance metric. The algorithm typically incorporates knowledge about various system parameters, such as processor speed, cache sizes, and cache line migration costs. It furthermore tries to perform a reasonable prediction about future workloads incorporating previous workload patterns.



(a) Five virtual machines running on four physical processors.

(b) Processing time for each processor is apportioned evenly among all virtual machines running on that processor.

(c) Balloon processes are started to even out the differences in processor execution speed as seen by the virtual machines. Rather than consuming CPU cycles, the balloon processes donate their CPU time to other virtual machines.

Figure 5. Time ballooning

By multiplexing multiple virtual processors on one physical CPU the virtualization layer modifies the fundamental system parameters which are used to derive the process distribution, e.g., linear time, identical processor speeds and assumptions about cache working sets due to scheduling order. Based on its wrong assumptions the guest OS's scheduling algorithm will distribute processes among physical processors in a sub-optimal way leading to over-commitment or under-utilization

To perform correct scheduling decisions the guest scheduler has to be made aware of the virtualization layer and incorporate the additional system parameters. Instead of distributing load equally between all processors it should distribute based on the percentage of physical resource allocation.

Using ideas similar to memory ballooning in the VMware ESX Server [25], we propose a new mechanism, *time ballooning*, that enables the hypervisor to partake in the load balancing decisions without requiring modification of the scheduling code itself. A balloon module is loaded into the guest OS as a pseudo-device driver. Periodically, the balloon module polls the virtualization layer to determine how much processing time the virtual machine is allotted on the different CPUs. It then generates virtual workload making the guest OS believe that it is busy executing a process during the time of no physical CPU allocation. The virtual workload levels out imbalances of physical processor allocation (see Figure 5 (c)). The time balloon is used to correct the scheduler's assumption that all processors have the same processing speed, leading to the anticipated distribution.

The method for generating virtual workload depends on the specific load balancing scheme of the guest OS and cannot be generalized. We investigated two Linux scheduling algorithms, the latest, better-scalable O(1) scheduler and the original, sampling-based work stealing algorithm. Since our focus was on non-intrusive solutions, we explicitly avoided modifications of the guest OS' scheduling code that may have resulted in a more efficient solution.

4.1 Linux O(1) Scheduler

Linux's O(1) scheduler uses processor local run queues. Linux bases load balancing decisions on two parameters, the run queue length, using the minimum of two sampling points, and cache working-set estimates based on last execution time. When the balancing algorithm finds a length difference of more than 25% between the current and the busiest run queue, or when the current processor falls idle, it starts migrating processes until the imbalance falls below the 25% threshold.

To reflect differences in virtual machine CPU resource allocations, the balloon module has to manipulate the run queue length. This can be achieved non-intrusively by generating low priority virtual processes with a fixed CPU affinity. In the balloon module we calculate the optimal load distribution using the total number of ready-to-run processes and the allocated CPU share for each virtual CPU. Based on the calculated optimal distribution we add balloon processes to each run queue until all have equal length to the longest run queue, i.e., the virtual CPU with the largest physical processor allocation.

For a guest OS with n virtual CPUs, a total of p running processes, and a CPU specific processor share s_{cpu}, the number of balloon processes b on a particular virtual processor is

$$b_{cpu} = \left\lceil \frac{\max(s) - s_{cpu}}{\sum_{i=1}^{n} s_i} \cdot p \right\rceil$$

Rounding up to the next full balloon results in at least one balloon process on all but those virtual CPUs with the largest s_{cpu} and thereby avoids aggressive rebalancing towards slower virtual CPUs that fall idle.

4.2 Linux Work-Stealing Scheduler

The second scheduling algorithm, the work-stealing algorithm, migrates processes under two conditions. The first condition responds to newly runnable processes. When a processor P_1 must schedule a newly woken task, it can suggest to idle processor P_2 to acquire the task. The migration completes only when P_2 chooses to execute the task. In the second migration condition, which takes place during general scheduling events such as end-of-timeslice, a processor can choose to steal any process from the centralized task list which is not hard-bound to another processor.

Linux bases load balancing decisions purely on the characteristics of a process, calculated as a "goodness" factor. The decision to steal a process is independent of the status of other processors, and thus doesn't consider factors such as the number of processes associated with other processors.

To influence load distribution, a balloon module has to give the impression that while the virtual processor is preempted it is not idle (i.e., it is executing a virtual process), to avoid migration attempts from other processors. The module could add balloon threads to the task list, bound to a particular processor, and which yield the virtual machine to the hypervisor when scheduled. But the module is unable to guarantee that Linux will schedule the balloon threads at appropriate times. The likelihood of scheduling balloon tasks can be increased by adjusting their priorities.

An alternative to balloon processes is possible. The work-stealing algorithm stores the inventory of running tasks in a central list, and thus if these tasks possess the property of cache affinity, then their task structures are likely to possess a field to represent preferred processor affinity (as is the case for Linux). The balloon module can periodically calculate an ideal load distribution plan, and update the tasks' preferred processor. Thus, as processors perform scheduling decisions, they'll find jobs in the task list biased according to feedback from the hypervisor.

5 Evaluation

5.1 Virtualization Architecture

For our experiments we used a paravirtualization approach running a modified Linux 2.4.21 kernel on top of a microkernel-based hypervisor [13, 23]. With our approach the management and device access parts of the hypervisor run unprivileged in user-space; interaction with the virtual machine manager, including device access, takes place via the microkernel's inter-process communication (IPC) facility.

We modified the Linux kernel to utilize the hypervisor's virtual memory management and scheduling primitives. All device access was wrapped into virtual device drivers that communicate with the real device drivers in the hypervisor.

The hypervisor supports all core device classes: hard disk, Gigabit Ethernet, and text console. Furthermore, it manages memory and time allocation for all virtual machines. To reduce device virtualization overhead we export optimized device interfaces using shared memory communication and IPC.

Currently, the hypervisor partitions memory and processors statically, i.e., no paging of virtual machines is taking place and virtual processors do not migrate between physical CPUs.

5.2 Lock Modeling

Our paravirtualization approach permitted us to reimplement the Linux kernel locks to study the benefits of intrusive lock-holder preemption avoidance. We implemented delayed preemption locks, pessimistic yield locks, and optimistic yield locks (with brief spinning).

The delayed preemption locks were constructed to inhibit preemption whenever at least one lock was held. Each virtual CPU (VCPU) maintained a count of acquired locks. Upon acquiring its first lock, a VCPU enabled its delayed preemption flag to prevent preemption by the hypervisor. The flag was cleared only after all locks were released. Setting and clearing the flag was a low cost operation, and only involved manipulating a bit in a page shared between the hypervisor and the VCPU. If the hypervisor signaled that it actually delayed a preemption, via another bit in the shared page, then the Linux kernel would voluntarily release its time slice immediately after releasing its last lock.

The yield locks were pessimistic and assumed that any spinning was due to a preempted lock, thereby immediately yielding the time slice if unable to acquire the lock. We also used an optimistic yield lock, which first spun on the lock for 20μs (as suggested from lock holding times in Figure 2), and then yielded the time slice with the assumption that the lock was preempted.

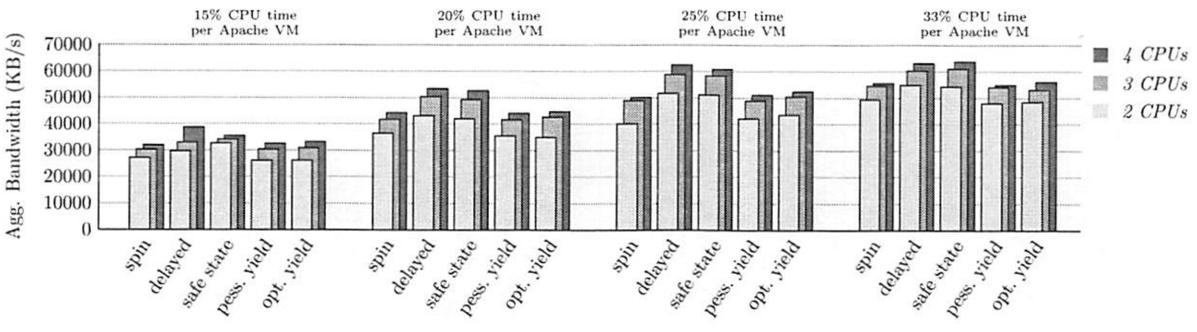


Figure 6. Bandwidth measurement for Apache 2 benchmark. Two virtual machines are configured to run on 2, 3 and 4 CPUs. For each CPU configuration the virtual machines are further configured to consume 15%, 20%, 25% or 33% of the total CPU time. A third virtual machine with a CPU intensive workload consumes the remaining CPU time. The bars show the aggregate bandwidth of the two VMs running the web servers.

To model the performance of lock-holder preemption avoidance with a fully virtualized virtual machine, and thus an unmodified guest OS, we had to create estimates using our paravirtualization approach. Given that paravirtualization can outperform a fully virtualized VM [3], our results only estimate the real benefits of lock preemption avoidance in a fully virtualized VM. Our safe-state implementation precisely models the behavior we describe in Section 3.2. We model safe-state detection by using the hypervisor's delayed preemption flag. When the guest OS executes kernel code we enable the delayed preemption flag, thus compelling the hypervisor to avoid preemption at the end of normal time slice. After finishing kernel activity (upon resuming user code or entering the idle loop), we clear the preemption flag. If Linux kernel code exceeds a one millisecond preemption grace period, it is preempted.

We observed spin-lock performance by using standard Linux spin-locks. The spin-lock data apply to the case of using spin-locks for paravirtualization, and they approximate the case of a fully virtualized VM.

5.3 Execution Environment

Experiments were performed with a Dell PowerEdge 6400, configured with four Pentium III 700 MHz Xeon processors, and an Intel Gigabit Ethernet card (using an 82540 controller). Memory was statically allocated, with 256 MB for the hypervisor, and 256 MB per virtual machine.

The guest operating system was a minimal Debian 3.0, based on a modified Linux 2.4.21 kernel. Most Debian services were disabled.

The hard disk was unused. Instead, all Linux instances utilized a 64 MB RAM disk. Linux 2.4.21 intelligently integrates the RAM disk with the buffer cache which makes this setup comparable to a hot buffer cache scenario.

5.4 Synthesized Web Benchmark

The synthesized web benchmark was crafted to tax the Linux network and VFS subsystems, in effect, to emulate a web server under stress. It used Apache 2, for its support of the Linux sendfile system call. The sendfile method not only offloads application processing to the kernel, but it also supports network device hardware acceleration for outbound checksum calculations and packet linearization.

The benchmark utilized two virtual machines, each hosting a web server to serve static files. The two virtual machines competed for the network device, and each had an equal amount of processor time. A third virtual machine provided an adjustable processor load, to absorb processor time.

5.5 Lock-Holder Preemption Avoidance Data

For studying the effect of our lock-holder preemption schemes, we instrumented Linux's synchronization primitives, and measured the throughput of the synthetic web benchmark described in the previous section.

To capture lock scaling in terms of number of processors, the Linux instances were configured to activate two, three, or four processors of our test system. Further, for each processor configuration we configured the virtual machines hosting the web servers to consume 15%, 20%, 25% or 33% of the total CPU time. Figure 6 summarizes our results, showing the aggregate bandwidth of both web servers for the different virtual machine and locking scheme configurations.

The results often reveal a substantial performance difference between the various locking techniques. In order

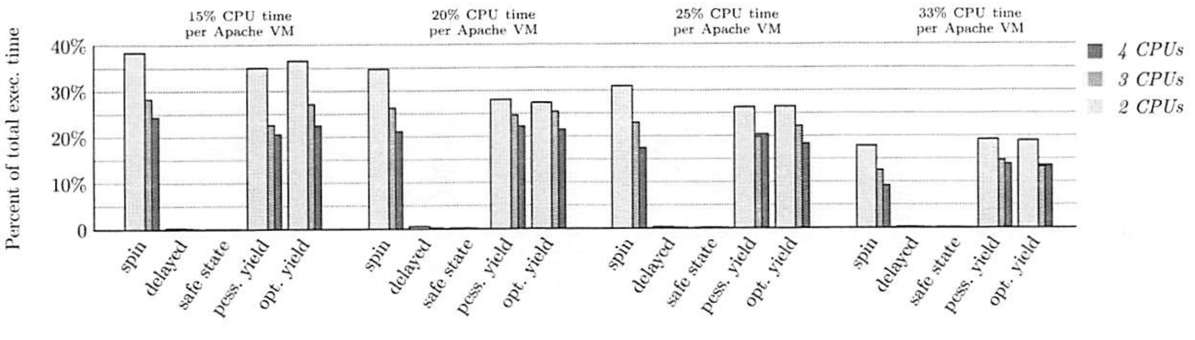


Figure 7. Extended lock-holding times for various virtual machine configurations (same configurations as in Figure 6). The bars show the extended lock-holding time for one of the web server VMs, per processor, expressed as a percentage of the run time. An extended lock-hold is one which exceeds 1 ms.

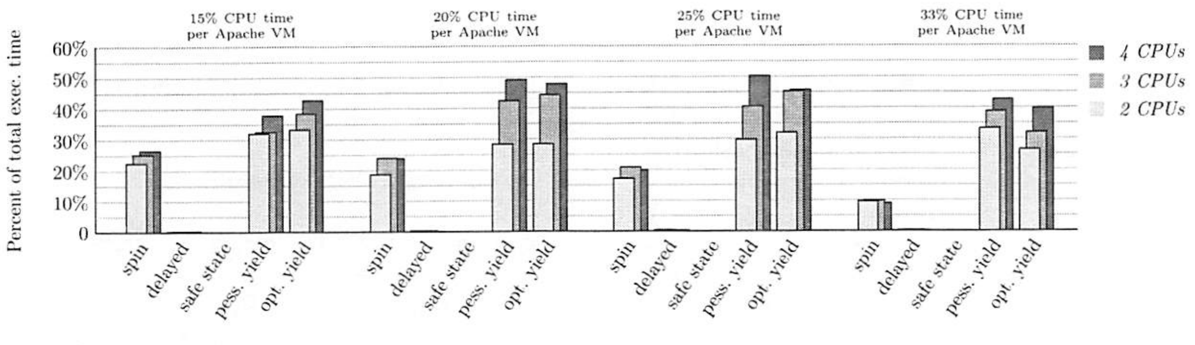


Figure 8. Extended lock-wait times for various virtual machine configurations (same configurations as in Figure 6). The bars show the extended time spent waiting for locks on one of the web server VMs, per processor, expressed as a percentage of the run time. An extended lock-wait is one which exceeds 1 ms.

to explain these differences, we measured lock holding times in each experiment. More precisely, we measured the amount of wall-clock time that a lock was held by the virtual machine running one of the web servers. Figure 7 shows the extended lock-holding time as a percentage of the virtual machine's real execution time during the benchmark. To distinguish between normal locking activity and lock-preemption activity, we show only extended lock-holding time. A lock-hold time is considered extended if it exceeds one millisecond.

We also measured the total time spent acquiring locks in each of the experiments (i.e., the lock spin-time or wait-time). These lock-acquire times are presented in Figure 8. Again, the time is measured relative to the real execution time of the benchmark. The data include only wait-times which exceeded one millisecond, to distinguish between normal behavior and lock-wait activity due to preempted locks.

5.6 Time Ballooning

Figure 9 presents the result of an experiment designed to show the effectiveness of the O(1) time ballooning algorithm with asymmetric, static processor allocation. We ran two virtual machines, A and B, each configured to run on two physical processors. The goal was to fairly distribute physical processor time between all processes. Both VMs were running a Linux 2.4.21 kernel with the O(1) scheduler patch and our balloon module. Virtual machine A was configured with 30% of the processing time and ran ten CPU intensive tasks. Virtual machine B was configured with the remaining 70% of the processing time, but only ran one CPU intensive task for a short period of time; the other CPU was idle.

Before VM B started using its processing time, VM A used all processing time on both CPUs. Once VM B started processing (at about 7 seconds into the experiment), virtual machine A's processing share on CPU 0 immediately dropped to 30%. Shortly thereafter, VM A detected the imbalance in processing time, and attempted to mitigate the problem by inserting balloon

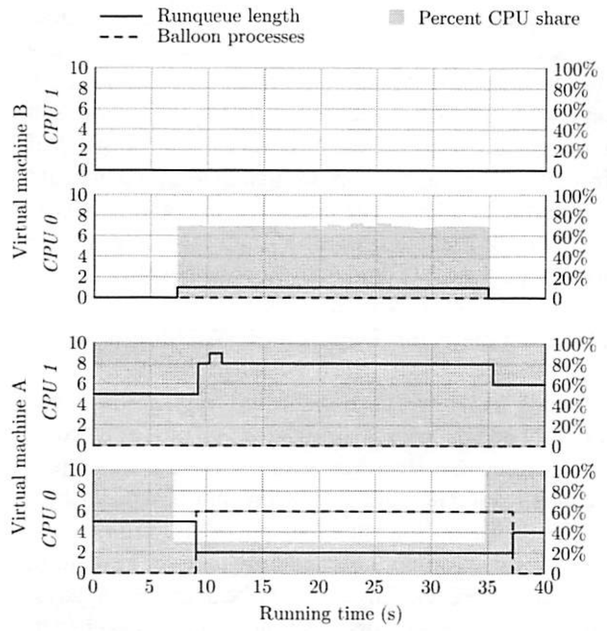


Figure 9. Time ballooning induced load balancing.

processes on the slower CPU. This in turn caused Linux to start migrating processes to the other CPU.

When virtual machine B ceased using its processing time (at about 35 seconds into the experiment), virtual machine A again got the full CPU share, causing the balloon processes to be removed, and Linux to perform another load-balancing.

6 Discussion and Future Work

6.1 Lock-holder Preemption Behavior

The web server benchmark provided complicated system dynamics. The processor utilization at almost every data point fell below the allocated processor share (we used a 5 ms time slice), suggesting that the workload was I/O-bound. On the other hand, the bandwidth is (non-linearly) proportional to the amount of processor share, suggesting a processor-bound workload. The sequencing of virtual machine scheduling, compared to packet arrival and transmit periods, probably causes this behavior. When a guest OS has access to the physical processor it won't necessarily be saturated with network events, since the network card may be sending and receiving packets for a competing virtual machine. Other factors can also regulate web server performance. For example, when a virtual machine sees little concurrent access to both the network device and the processor, the role of kernel buffers and queues becomes important. The buffers and queues must hold data until a resource is available, but buffer overruns lead to work underflow, and thus under-utilization of the time slice. The sequencing of events may also lead to high packet processing latency, causing TCP/IP packet retransmits, which increases the load on the limited resources.

During the experiments involving our lock-holder preemption avoidance techniques we set the size of the preemption window (the value of w in Section 3.3) to 1 ms. Since more than 96% of the unsafe times for an Apache 2 workload fall below the 1 ms boundary (see Figure 3), the non-intrusive technique did not suffer from excessive amounts of lock-holder preemptions. As such, the web-server throughput achieved with the non-intrusive technique was on par with the intrusive technique.

The hypervisor uses a proportional share stride-scheduling algorithm [26, 27], which ensures that every virtual machine receives its allocated share of the processor. However, the scheduler doesn't impose a ceiling on a virtual machine's processor utilization. If one virtual machine under-utilizes its processor share for a given unit of time, then another virtual machine can pilfer the unused resource, thus increasing its share. With I/O bound workloads, we witness processor stealing (because we have time slice under-utilization). In essence, the scheduling algorithm permits virtual machines to rearrange their time slices. The concept of time-stealing also applies to yield-style locks. When a virtual processor yields its time slice, rather than wait on a lock-acquire, it may avoid many milliseconds of spinning. Yielding the time slice, though, enables the hypervisor to schedule another virtual machine sooner, and thus return to scheduling the yielding virtual machine earlier, at which point the lock should be released from the original lock holder. Time slice yielding, as an alternative to spinning on preempted locks, will improve the overall efficiency of the system, as it avoids processor waste. And via the rearrangement of time slices, yield-style locks may improve their processor/packet sequencing, and improve performance.

Increasing parallelism by adding more processors increases the likelihood of lock contention. Thus preemption of a lock holder can lead to spinning by more processors. As Figure 8 shows, the lock-wait time increases with the number of processors.

In the graphs which represent lock-holding time (Figure 7) and lock-waiting time (Figure 8), the mechanisms which avoid lock-holder preemption (delayed-preemption locks and safe-state detection) have under 1% time involved with locking activity. In contradistinction, the spin-locks lead to severe locking times and spinning times due to preempted locks. The yield-style locks are susceptible to lock preemption, as they only focus on recovering from lock spinning, and also suffer from long lock holding times. And due to releasing their time slices for use by other virtual machines, yield-style locks suffer from long lock-wait times as well. The tactic of avoiding spin-time by yielding doesn't help the benchmark results, and often leads to performance worse than spin-locks. Spin-locks may spin for a long time, but they have a chance of acquiring the lock before the end of time slice, and thus to continue using the time slice.

6.2 Time Ballooning

Our time ballooning mechanism is targeted towards processor allocations with infrequent changes. Highly dynamic processor reconfigurations introduce load-dependent properties, and are a topic of future work. To support dynamic load balancing in response to adjustments of the processor allocations requires attention to several variables, such as the rates at which the guest OS can rebalance and migrate tasks, allocation sampling rates, sampling overhead, cache migration costs, sample history window size, allocation prediction, and attenuation of thrashing. Likewise, response to burst loads is another dynamic situation for future work. For example, to achieve optimal use of the available physical processor resources, web servers distributed across asymmetric processors may require load balancing of incoming wake-up requests, or load balancing of the web server processes after they wake/spawn.

The time ballooning mechanism is designed to be installed in unmodified guest operating systems via device drivers. Where it is possible to modify the guest operating system, one can construct feedback scheduling algorithms optimized for the OS and workload and virtualization environment.

6.3 Dealing with Lock-Holder Preemption

An alternative to avoiding preemption of lock holders in the first place is to deal with the effects of the preemption: Have the hypervisor detect extensive spinning and schedule other, more useful work instead.

Techniques to detect spinning include instruction pointer sampling and trapping on instructions that are used in the back-off code of spin-locks. Descheduling a virtual CPU immediately after a failed acquire operation is expected to show the same behavior as pessimistic yield locks. To reduce the yield time, one could look for a lock release operation following a failed lock acquire. Lock release operations can be detected by write-protecting pages containing a lock, using debug registers to observe write accesses, or, with additional hardware support, through extensions of the cache snooping mechanism.

However, preempting a virtual CPU due to a remote release operation on a monitored lock may preempt another lock holder. Most likely, release operations on a contended lock occur with much higher frequency than preemptions due to regular VM scheduling. Instead of offering a solution, chances are that this approach could amplify the problem. The potential costs and complexity of detecting spinning on a lock and executing another virtual CPU instead could outweigh the benefits.

6.4 Coscheduling Dependent Workloads

In this paper we have discussed application workloads that do not possess strong cross-processor scheduling requirements. However, some workloads in the parallel application domain do rely on spin-based synchronization barriers or application spin-locks, and thus necessitate some form of coscheduling in order to perform efficiently. Coscheduling can only be achieved on physical resources (processors and time), and the coscheduling requirements in the guest OS must therefore be communicated to the virtualization layer so that they can be processed on a physical processor basis.

Making the virtualization layer aware of an application's coscheduling requirements and devising algorithms for fulfilling these requirements is future work.

7 Related Work

The problems that may result from preempting parts of a parallel application while holding a lock are well-known [16, 20, 30] and have been addressed by several researchers [1, 4, 8, 17, 29]. Proposed solutions include variants of scheduler-aware synchronization mechanisms and require kernel extensions to share scheduling information between the kernel's scheduler and the applications. This translates well to our para-virtualization approach where the guest OS kernel provides information to the hypervisor. To our knowledge no prior work has applied this technique in the context of virtual machines and their guest OSs.

Only very few virtual machine environments offer multiprocessor virtual machines, and they either avoid lock-holder preemption completely through strict use of gang scheduling [5, 10], use gang scheduling whenever they see fit [24], or they choose flexible scheduling but don't address lock-holder preemption [15].

Load balancing across non-uniformly clocked, but otherwise identical CPUs is a standard technique in cluster systems. However, we know of no commodity operating system for tightly coupled multiprocessors that would explicitly support such a configuration. The few existing virtual multiprocessor VM environments either implicitly enforce equal virtual CPU speeds (through gang scheduling) or do not consider speed imbalances. As such, we know of no prior art where a virtual machine environment would coerce its guest operating system to adapt to changing, or at least statically different, virtual CPU speeds. Our solution was inspired by the ballooning technique for reclaiming memory from a guest OS [25].

Performance isolation is a quite well-researched principle [2, 3, 12, 18] in the server consolidation field. For CPU time our hypervisor enforces resource isolation using a proportional share stride-scheduling algorithm [26, 27].

8 Conclusion

Virtual machine based server consolidation on top of multiprocessor systems promises great flexibility with respect to application workloads, while providing strong performance isolation guarantees among the consolidated servers. However, coupling virtual machine environments with multiprocessor systems raises a number of problems that we have addressed in this paper.

First, our schemes for avoiding preemption of lock-holders in the guest operating systems prevents excessive spinning times on kernel locks, resulting in noticeable performance improvements for workloads exhibiting high locking activity. Our lock-holder preemption avoidance techniques are applicable to both paravirtualized and fully virtualized systems.

Second, the allocation of physical processing time to virtual machines can result in virtual machines experiencing asymmetric and varying CPU speeds. Our time ballooning technique solves the problem by creating artificial load on slower CPUs, causing the guest OS to migrate processes to the CPUs with more processing time.

Combined, our solutions enable scalable multiprocessor performance with flexible virtual processor scheduling.

Acknowledgements

We would like to thank Åge Kvalnes for his insightful feedback. Further thanks to the anonymous reviewers for their valuable comments. The work presented in this paper was in part funded through a grant by Intel Corporation.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 22–25 1999.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 19–22 2003.

[4] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the 16th Symposium on Operating Systems Principles*, Saint Malo, France, October 5–8 1997.

[6] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 14–17 1994.

[7] Connectix. *Virtual PC for Windows, Version 5.1, User Guide*, second edition, July 2002.

[8] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proc. of the USENIX Workshop on Unix and Supercomputers*, Pittsburg, PA, September 26–27 1988.

[9] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[10] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[11] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 28–31 1996.

[12] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proc. of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 23–26 2003.

[13] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symposium on Operating System Principles*, Saint Malo, France, October 5–8 1997.

[14] Michael Hohmuth and Hermann Härtig. Pragmatic non-blocking synchronization for Real-Time systems. In *Proc. of the 2001 USENIX Annual Technical Conference*, June 25–30 2001.

[15] IBM. *z/VM Performance, Version 4 Release 4.0*, fourth edition, August 2003.

[16] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 13–16 1991.

[17] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, 1997.

[18] Åge Kvalnes, Dag Johansen, Robbert van Renesse, and Audun Arnesen. Vortex: an event-driven multiprocessor operating system supporting performance isolation. Technical Report 2003-45, University of Tromsø, 2003.

[19] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY, 1991.

[20] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proc. of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1–5 1997.

[21] John. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of the 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, FL, October 18–22 1982.

[22] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25–30 2001.

[23] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, Universität Karlsruhe, May 1 2003.

[24] VMware Inc. VMware ESX Server online documentation. http://vmware.com/, 2003.

[25] Carl A. Waldsburger. Memory resource management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 9–11 2002.

[26] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 14–17 1994.

[27] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, 1995.

[28] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 9–11 2002.

[29] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 19–21 1995.

[30] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

Using Hardware Performance Monitors to Understand the Behavior of Java Applications

Peter F. Sweeney[1] Matthias Hauswirth[2*]
Brendon Cahoon[1] Perry Cheng[1] Amer Diwan[2] David Grove[1] Michael Hind[1]
[1]*IBM Thomas J. Watson Research Center*
[2]*University of Colorado at Boulder*

Abstract

Modern Java programs, such as middleware and application servers, include many complex software components. Improving the performance of these Java applications requires a better understanding of the interactions between the application, virtual machine, operating system, and architecture. Hardware performance monitors, which are available on most modern processors, provide facilities to obtain detailed performance measurements of long-running applications in real time. However, interpreting the data collected using hardware performance monitors is difficult because of the low-level nature of the data.

We have developed a system, consisting of two components, to alleviate the difficulty of interpreting results obtained using hardware performance monitors. The first component is an enhanced VM that generates traces of hardware performance monitor values while executing Java programs. This enhanced VM generates a separate trace for each Java thread and CPU combination and thus provides accurate results in a multithreaded and multiprocessor environment. The second component is a tool that allows users to interactively explore the traces using a graphical interface. We implemented our tools in the context of Jikes RVM, an open source Java VM, and evaluated it on a POWER4 multiprocessor. We demonstrate that our system is effective in uncovering as yet unknown performance characteristics and is a first step in exploring the reasons behind observed behavior of a Java program.

1 Introduction

Modern microprocessors provide hardware performance monitors (HPMs) to help programmers understand the low-level behavior of their applications. By counting the occurrences of events, such as pipeline stalls or cache misses, HPMs provide information that would otherwise require detailed and, therefore, slow simulations. Because the information provided by HPMs is low-level in nature, programmers still have the difficult task of determining how the information relates to their program at the source level. This paper describes a system that alleviates some of this difficulty by relating HPM data to Java threads in a symmetric multiprocessor environment (SMP).

Our system overcomes the following four challenges in interpreting HPM data for Java programs. First, because a Java virtual machine's rich runtime support uses the same hardware resources as the application, resource usage of the VM threads needs to be distinguished from those of the application. Second, because Java applications often employ multiple threads, each thread's resource usage needs to be distinguished. Third, because the characteristics of even a single Java thread may vary during its execution it is important to capture the time-varying behavior of a thread. Fourth, because in a SMP environment a single Java thread may migrate among several physical processors, the performance characteristics of each thread and CPU combination need to be attributed correctly.

Our system consists of two components: an enhanced VM that generates traces of hardware events and a visualization tool that processes these traces. The enhanced VM, an extension to Jikes RVM (Research Virtual Machine), accesses the PowerPC HPMs to generate a trace from a running application. The trace consists of a sequence of trace records that capture hardware performance events during the length of a thread scheduler quantum for each processor. In addition to calculating aggregate metrics, such as overall IPC (instruction per cycle) for each thread, these traces allow one to explore the time-varying behavior of threads, both at application and VM level.

The output of the trace generator is too large to be immediately usable. For example, one thirty-second run has almost sixty thousand events in its trace. The visualization tool allows users to interactively explore the traces and to compare multiple metrics (e.g., cache misses and memory stalls) graphically and side-by-side. In this way a user can explore hypotheses interactively (e.g., are metrics A and B correlated?).

We demonstrate the usefulness of the system by applying it

*Work done while a summer student at IBM Research in 2003.

to a variation of the SPECjbb2000benchmark. We show that the system is effective in identifying performance anomalies and also helps us to explore their cause.

To summarize, our contributions are as follows:

- We describe an extension to Jikes RVM to access PowerPC HPMs and accurately attribute them to Java threads in a SMP environment. Although many prior systems (such as DCPI [5] and OProfile [26]) give users access to HPMs, we believe ours is the first system that generates Java thread-specific traces over time of HPM data for multithreaded applications on a multiprocessor.
- We present the `Performance Explorer`, a visualization tool for interactively and graphically understanding the data.
- We use our tools to identify performance anomalies in our benchmark and also to explore the cause of the anomalies. Explaining the cause of the anomalies was tricky due to the complexity of both the software (which uses hard to understand features such as adaptive compilation and garbage collection) and the hardware (which employs an elaborate memory system design). Our visualization tool was invaluable in this exploration, but we realize that there is still more work to be done in this area.

The rest of this paper is organized as follows. Section 2 provides the background for this work, including an overview of Jikes RVM and the existing mechanism for accessing the PowerPC HPMs under AIX. Section 3 describes the design and implementation of the VM extension mechanism for recording HPMs. Section 4 introduces the visualization tool. Section 5 illustrates how the tool can be used to help understand the hardware performance of Java applications. Section 6 discusses related work. Section 7 outlines avenues for future work and Section 8 draws some conclusions.

2 Background

This section provides the background for this work. Section 2.1 provides an overview of the existing Jikes RVM infrastructure that this paper uses as a foundation. Section 2.2 summarizes hardware performance monitors on AIX.

2.1 Jikes RVM

Jikes RVM [19] is an open source research virtual machine that provides a flexible testbed for prototyping virtual machine technology. It executes Java bytecodes and runs on the Linux/IA32, AIX/PowerPC, Linux/PowerPC platforms, and OS X/PowerPC platforms. This section briefly provides background on a few relevant aspects of Jikes RVM. More details are available at the project web site [19] and in survey papers [2, 12, 6].

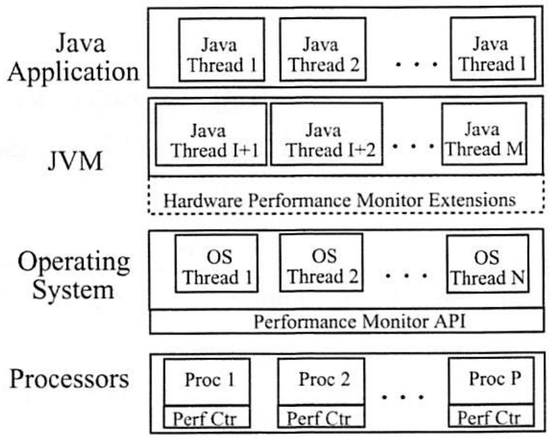


Figure 1: Architectural overview.

Jikes RVM is implemented in the Java programming language [3] and uses Java threads to implement several subsystems, such as the garbage collector [10] and adaptive optimization system [6]. Thus, our HPM infrastructure provides insight both into the performance characteristics of Java programs that execute on top of Jikes RVM and into the inner workings of the virtual machine itself. In particular, by gathering per-Java-thread HPM information the behavior of these VM threads is separated from application threads. However, VM services, such as memory allocation, that execute as part of the application thread are not separately distinguished.

As shown in Figure 1, Jikes RVM's thread scheduler maps its M Java threads (application and VM) onto N Pthreads (user level POSIX threads). There is a 1-to-1 mapping from Pthreads to OS kernel threads. A command line argument to Jikes RVM specifies the number of Pthreads, and corresponding kernel threads, that Jikes RVM creates. The operating system schedules the kernel threads on available processors. Typically Jikes RVM creates a small number of Pthreads (on the order of one per physical processor). Each Pthread is called a *virtual processors* because it represents an execution resource that the virtual machine can use to execute Java threads.

To implement M-to-N threading, Jikes RVM uses compiler-supported quasi-preemptive scheduling by having the two compilers (baseline and optimizing) insert *yieldpoints* into method prologues, epilogues, and loop heads. The yieldpoint code sequence checks a flag on the virtual processor object; if the flag is set, then the yieldpoint invokes Jikes RVM's thread scheduler. Thus, to force one of the currently executing Java threads to stop running, the system sets this flag and waits for the thread to execute a yieldpoint sequence. The flag can be set by a timer interrupt handler (signifying that the 10ms scheduling quantum has expired) or by some other system service (for example, the need to initiate a garbage collection) that needs to preempt the Java thread to schedule one of its own daemon threads. Because yieldpoints are a subset of the GC safe points, i.e., program points where the stack references (or GC roots) are known, only the running Java threads need to reach a GC safe point to begin a stop-the-world garbage collection; threads in the scheduler's ready queue are already prepared for a garbage collection.

2.2 AIX Hardware Performance Monitors

Most implementations of modern architectures (e.g. PowerPC POWER4, IA-64 Itanium) provide facilities to count hardware events. Examples of typical events that may be counted include processor cycles, instructions completed, and L1 cache misses. An architecture's implementation exposes a software interface to the event counters through a set of special purpose hardware registers. The software interface enables a programmer to monitor the performance of an application at the architectural level.

The AIX 5.1 operating system provides a library with an application programming interface to the hardware counters as an operating system kernel extension (pmapi).[1] The API, shown as part of the operating system layer in Figure 1, provides a set of system calls to initialize, start, stop, and read the hardware counters. The initialization function enables the programmer to specify a list of predefined events to count. The number and list of events depends on the architecture's implementation, and vary substantially between different PowerPC implementations (e.g. PowerPC 604e, POWER3, and POWER4). The API provides an interface to count the events for a single kernel thread or for a group of threads. The library automatically handles hardware counter overflows and kernel thread context switches.

A programmer can count the number of times a hardware event occurs in a code segment by manually instrumenting a program with the appropriate API calls. Prior to the code segment, the instrumentation calls the API routines to initialize the library to count certain events and to commence counting. After the code segment, the instrumentation calls the API routines to stop counting, read the hardware counter events, and optionally print the values. The HPM toolkit provides a command line facility to measure the complete execution of an application [15].

Some processors provide more sophisticated facilities to access HPM data. These facilities include thresholding and sampling mechanisms. The thresholding mechanism allows the programmer to specify a threshold value and an event. Only if the event exceeds the threshold value is the hardware counter incremented. The sampling mechanism allows the programmer to specify a value, n, and an event. When the event occurs for the nth time, the hardware exposes the executing instruction and operand address to the software. The POWER4 architecture provides thresholding and sampling capabilities, but the AIX 5.1 kernel extension library (pmapi) does not support them.

[1]The package is also available for earlier versions of AIX at IBM's AlphaWorks site `www.alphaworks.ibm.com/tech/pmapi`.

3 VM Extensions

This section describes extensions to Jikes RVM that enable the collection of trace files containing thread-specific, temporally fine-grained HPM data in an SMP environment. The section begins by enumerating the design goals for the infrastructure. It then describes the trace record format and highlights some of the key ideas of the implementation. Finally, it considers issues that may arise when attempting to implement similar functionality in other virtual machines.

3.1 Design Goals

There are four primary goals for the HPM tracing infrastructure.

Thread-specific data The infrastructure must be able to discriminate between the various Java threads that make up the application. Many large Java applications are multi-threaded, with different threads being assigned different portions of the overall computation.

Fine-grained temporal information The performance characteristics of a thread may vary over time. The infrastructure must enable the identification of such changes.

SMP Support The infrastructure must work on SMPs. In an SMP environment, multiple threads will be executing concurrently on different physical processors, and the same Java thread may execute on different processors over time. There will be a stream of HPM data associated with each virtual processor and it must be possible to combine these separate streams into a single stream that accurately reflects the program's execution.

Low overhead Low overhead is desirable both to minimize the perturbations in the application introduced by gathering the data and to enable it to be used to gather traces in production environments or for long-running applications.

3.2 Trace Files

When the infrastructure is enabled, it generates a trace file for each Jikes RVM virtual processor, and one meta file. This section describes the structure of the trace and meta files, and discusses how the data gathered enables the system to meet the design goals enumerated in Section 3.1.

The core of the trace file is a series of trace records. Each trace record represents a measurement period in which exactly one Java thread was executing on the given virtual processor. The HPM counters are read at the beginning and end of the measurement period and the change in the counter values is reported in the trace record. A trace record contains the following data:

Virtual Processor ID This field contains the unique ID of the virtual processor that was executing during the measurement period. Although this information can be inferred (each trace file contains trace records from exactly one virtual processor), we chose to encode this in the trace record to simplify merging multiple trace files into a single file that represents an SMP execution trace.

Thread ID This field contains the unique ID of the thread that was executing during the measurement period.

Thread Yield Status This boolean field captures if the thread yielded before its scheduling quantum expired. It is implemented by negating the thread ID.

Top Method ID This field contains the unique ID of the top method on the runtime stack at the end of the measurement period, excluding the scheduler method taking the sample. Because the complete stack is available, this field could be extended to capture more methods as was done to guide profile-directed inlining [18].

Real Time This field contains the value of the PowerPC time base register at the beginning of the measurement period represented by this trace record. The time base register contains a 64-bit unsigned quantity that is incremented periodically (at an implementation-defined interval) [22].

Real Time Duration This field contains the duration of the measurement period as measured by the difference in the time base register between the start and end of the measurement period.

Hardware Counter Values These fields contain the change in each hardware counter value during the measurement period. The number of hardware counters varies among different implementations of the PowerPC architecture. In most anticipated uses, one of the counters will be counting cycles executed, but this is not required by the infrastructure.

As each trace record contains hardware counter values for a single Java thread on a single virtual processor, we are able to gather thread-specific HPM data. The key element for SMP support is the inclusion in the trace record of the real time values read from the PowerPC time base register. The primary use of the real time value is to merge together multiple trace files by sorting the trace records in the combined trace by the real time values to create a single trace that accurately models concurrent events on multiple processors. A secondary use of this data is to detect that the OS has scheduled other Pthreads on the processor during the measurement interval. Large discrepancies between the real time delta and the executed cycles as reported by the hardware counter indicate that some OS scheduling activity has occurred and that the Pthread shared the processor during the measurement period. Recall that the OS extension already distinguishes counters for each OS kernel thread.

In addition to trace records containing hardware counter values, a trace file may also contain marker records. We provide a mechanism, via calls to the VM, for a Java thread to specify program points that when executed will place a marker record in the trace file. These marker trace records, which can contain an arbitrary string, allow the programmer to focus on particular portions of an execution's trace. Because this mechanism is available at the Java level, it can be used to filter both application and VM activities, such as the various stages of garbage collection.

A meta file is generated in conjunction with a benchmark's trace files. The meta file specifies the number of HPM counter values in each trace record and provides the following mappings: from counter to event name, from thread ID to thread name, and from method ID to method signature. The number of counters and the mappings provide a mechanism to interpret a trace record, reducing a trace record's size by eliminating the need to name trace record items in each individual trace record.

3.3 Implementation Details

To enable Jikes RVM to access the C pmapi API we defined a Java class with a set of native methods that mirrors the functionality of the pmapi interface, represented by the dashed box of the JVM in Figure 1. In addition to enabling our VM extensions in Jikes RVM to access these functions, the interface class can also be used to manually instrument arbitrary Java applications to gather aggregate HPM data. We are using this facility to compare the performance characteristics of Java applications when run on Jikes RVM and on other JVMs.

The main extension point in Jikes RVM was to add code in the thread scheduler's context switching sequence to read the hardware counters and real time clock on every context switch in the VM. This information is accumulated into summaries for each virtual processor and Java thread, and written into a per virtual processor trace record buffer. Each virtual processor has two dedicated 4K trace record buffers and a dedicated Java thread, called a *trace writer* thread, whose function is to transfer the contents of a full buffer to a file. Trace records for a virtual processor are written into an active buffer. When the buffer is full, the virtual processor signals the trace write thread and starts writing to the other buffer.

By alternating between two buffers, we continuously gather trace records with low overhead. By having a dedicated Java thread drain a full buffer, and thus, not suspend the current thread to perform this task, we avoid directly perturbing the behavior of the other threads in the system. This implementation also enables easy measurement of the overhead of writing the trace file because HPM data is gathered for all Java threads, including the threads that are writing the trace files. In our experiments 1.7% of all cycles are spent executing the trace writer threads. The overhead of reading the hardware counters and real time clock on every thread switch and storing the trace information into the buffer is in the measurement noise. Thus, the total overhead of the infrastructure is less than 2%.

Minor changes were also made in the boot sequence of the virtual machine, and in the code that creates virtual processors and Java threads to appropriately initialize data structures. The VM extensions have been available in Jikes RVM [19] since the 2.2.2 release (June 2003).

The disk space required to store trace records is a function of the trace record size, the frequency of thread switches, and the number of virtual processors. For example, running one warehouse in the SPECjbb2000 benchmark on one virtual processor, we found that 12 Kbytes per second were written with a 10 millisecond scheduling quantum.

3.4 Discussion

Jikes RVM's M-to-N threading required an extension of the virtual machine to gather Java thread specific HPM data. In JVMs that directly map Java threads to Pthreads, it should be possible to gather *aggregate* Java thread specific HPM data using the pmapi library by making relatively simple extensions to read HPM counters when threads are created and terminated. So, in this respect the Jikes RVM implementation was more complex than it might have been in other JVMs. However, M-to-N threading made the gathering of fine-grained temporal HPM data fairly straightforward. A relatively simple extension to the context-switching sequence to read the HPM counters on every thread switch was sufficient to collect the desired data. Gathering this kind of data on virtual machines that do not employ M-to-N threading will probably be significantly more difficult because applying a similar design would require modifications to either the Pthread or OS thread libraries.

4 Visualization Tool

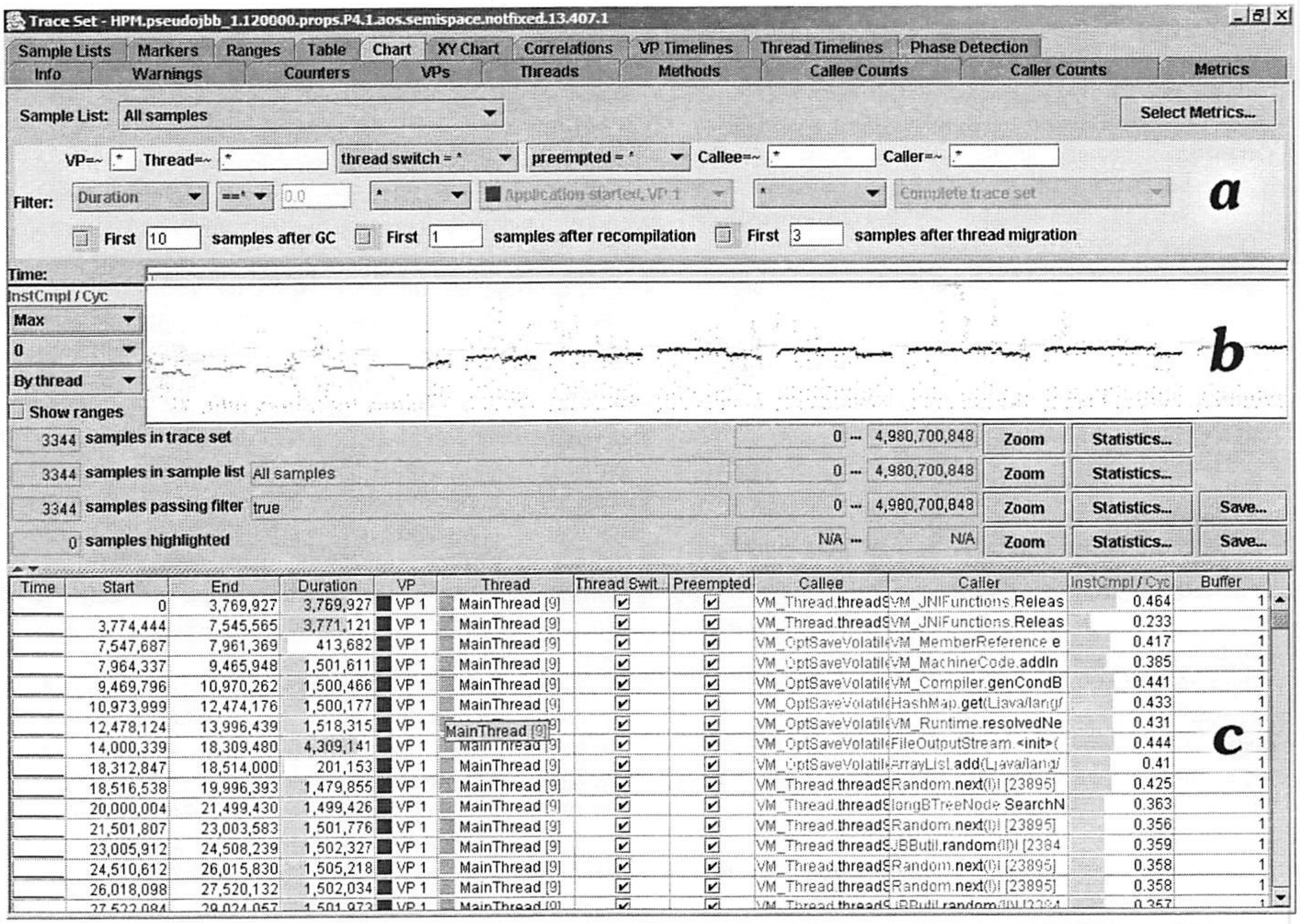
This section describes our visualization tool, called the `Performance Explorer`, which supports performance analysis through the interactive visualization of trace records that are generated by the VM extensions described in Section 3. Figure 2 presents an overview of the `Performance Explorer` when run with a variant of SPECjbb2000 using one warehouse on one virtual processor. The figure comprises three parts (`a`, `b`, and `c`), which are further described below.

The `Performance Explorer` is based on two key concepts: *trace record sets* and *metrics*. A *trace record set* is a set of trace records. Given the trace files of an application's execution, the `Performance Explorer` provides an initial trace record set containing all trace records of all threads on all virtual processors. It also provides filters to create subsets of a trace record set (e.g., all trace records of the MainThread, all trace records longer than 5 ms, all trace records with more than 1000 L1 D-cache misses, or all trace records ending in a Java method matching the regular expression "`.*lock.*`"). Part `a` of Figure 2 illustrates the user interface for configuring these filters. Furthermore, the `Performance Explorer` provides set operations (union, intersection, difference) on trace record sets.

A *metric* extracts or computes a value from a trace record. The `Performance Explorer` provides a metric for each hardware counter gathered in the trace files, and a metric for the trace record duration. The user can define new metrics using arithmetic operations on existing metrics. For example, instructions per cycle (IPC) can be computed using a computed metric that divides the instructions completed (INST_CMPL) event value by the cycles (CYC) event value. Part `b` in Figure 2 shows a graph for just one metric. The horizontal axis is wall-clock time, and the vertical axis is defined by the metric, which in this case is IPC. The vertical line that is a quarter of the way in from the left side of the graph represents a marker trace record generated by manual instrumentation of the program. This specific marker shows when the warehouse application thread starts executing. To the right of the marker, the applications enters a steady state where the warehouse thread is created and executed. To the left of the marker represents the program's start up where its main thread dominates execution. Each line segment in the graph (in this zoomed-out view of the graph most line segments appear as points) represents a trace record. The length of the line segment represents its wall clock duration. The color of the line segment (different shades of gray in this paper) indicates the corresponding Java thread. The user can zoom in and out and scroll the graph.

Part `c` of Figure 2 displays a table of the trace records visualized in the graph, one trace record per line. This table presents all the attributes of a trace record, including the values of all metrics plotted in the graph. Selecting a range of trace records in the graph selects the same records in the table, and vice versa. This allows the user to select anomalous patterns in the graph, for example the drop in IPC before each garbage collection, and immediately see all the attributes (like method names) of the corresponding trace records in the trace record table. The user can get simple descriptive statistics (sum, minimum, maximum, average, standard deviation, and mean delta) over the selected trace records for all metrics. Finally, a selection of trace records can be named and saved as a new trace record set.

In addition to providing time graphs and trace record tables, the `Performance Explorer` also provides several other ways to visualize the trace data. The `Performance Explorer` provides thread timelines, which are tables where each column represents a thread, each row represents an interval in time, and the cells are colored based on the processor that is executing the thread. The `Performance Explorer` provides processor timelines, where each column represents a processor, and the cells are colored based on the thread that executes on the processor. Cells in these timelines visualize a given metric (like IPC), and they are adorned with glyphs to show preemption or yielding of a thread at the


Trace Set - HPM.pseudojbb_1.120000.props.P4.1.aos.semispace.notfixed.13.407.1

Sample Lists | Markers | Ranges | Table | Chart | XY Chart | Correlations | VP Timelines | Thread Timelines | Phase Detection
Info | Warnings | Counters | VPs | Threads | Methods | Callee Counts | Caller Counts | Metrics

Sample List: All samples — Select Metrics...

VP=~ .* Thread=~ .* thread switch = * preempted = * Callee=~ .* Caller=~ .*

Filter: Duration ==* 0.0 * Application started, VP 1 * Complete trace set

First 10 samples after GC — First 1 samples after recompilation — First 3 samples after thread migration

Time:
InstCmpl / Cyc — Max — 0 — By thread — Show ranges

3344 samples in trace set — 0 ... 4,980,700,848 — Zoom — Statistics...
3344 samples in sample list All samples — 0 ... 4,980,700,848 — Zoom — Statistics...
3344 samples passing filter true — 0 ... 4,980,700,848 — Zoom — Statistics... — Save...
0 samples highlighted — N/A ... N/A — Zoom — Statistics... — Save...

Time	Start	End	Duration	VP	Thread	Thread Swit...	Preempted	Callee	Caller	InstCmpl / Cyc	Buffer
	0	3,769,927	3,769,927	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	VM_JNIFunctions.Releas	0.464	1
	3,774,444	7,545,565	3,771,121	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	VM_JNIFunctions.Releas	0.233	1
	7,547,687	7,961,369	413,682	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	VM_MemberReference.e	0.417	1
	7,964,337	9,465,948	1,501,611	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	VM_MachineCode.addIn	0.385	1
	9,469,796	10,970,262	1,500,466	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	VM_Compiler.genCondB	0.441	1
	10,973,999	12,474,176	1,500,177	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	HashMap.get(Ljava/lang/	0.433	1
	12,478,124	13,996,439	1,518,315	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	VM_Runtime.resolvedNe	0.431	1
	14,000,339	18,309,480	4,309,141	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	FileOutputStream.<init>(	0.444	1
	18,312,847	18,514,000	201,153	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatil	ArrayList.add(Ljava/lang/	0.41	1
	18,516,538	19,996,393	1,479,855	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.425	1
	20,000,004	21,499,430	1,499,426	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	longBTreeNode.SearchN	0.363	1
	21,501,807	23,003,583	1,501,776	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.356	1
	23,005,912	24,508,239	1,502,327	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	JBButil.random(II)I [2384	0.359	1
	24,510,612	26,015,830	1,505,218	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.358	1
	26,018,098	27,520,132	1,502,034	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.358	1
	27,522,084	29,024,057	1,501,973	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	JBButil.random(II)I [2384	0.357	1

Figure 2: Overview of the `Performance Explorer`.

end of a time slice. These timelines are helpful in analyzing scheduling effects. The `Performance Explorer` provides scatter plots, where the X and Y axes can be any given metric, and all trace records of a trace record set are represented as points. Lastly, the `Performance Explorer` calculates the correlations between any two metrics over a trace record set.

Due to the extensive number of HPM events that can be counted on POWER4 processors, and the limitation of a given event being available only in a limited number of hardware counters, the `Performance Explorer` provides functionality for exploring the available events and event groups.

Because of space considerations, subsequent figures only contain information from the `Performance Explorer` that is pertinent to the discussion at hand.

5 Experiments

This section demonstrates how we used the `Performance Explorer` to understand the performance behavior of a variant of the SPECjbb2000 benchmark on a PowerPC POWER4 machine.

Caches	Size	Line size	Kind
L1 Instr.	32KB	128B	direct mapped
L1 Data	64KB	128B	2-way set assoc.
L2 Unified	1.5MB	128B	8-way set assoc.
L3 local	32MB	512B	8-way set assoc.
L3 remote	32MB	512B	8-way set assoc.

Table 1: POWER4 three levels of cache hierarchy.

5.1 POWER4

The POWER4 [8] is a 64-bit microprocessor that contains two processors on each chip. Four chips can be arranged on a module to form an 8-way machine, and four modules can be combined to form a 32-way machine. The POWER4 contains three cache levels as described in Table 1. There is one L1 data and one L1 instruction cache for each core on a chip. The L1 caches are *store through* (write through); that is, a write to the L1 is stored through to L2. Two processors on a chip share an L2 cache. The L2 cache is inclusive of the two L1 data caches, but not inclusive of the two L1 instruction caches; that is, any data in the L1 data cache also resides in the L2 cache. However, data may reside in the L1 instruction cache that does not reside in the L2 cache. The L2 cache is *store in* (write back); that is, a write to L2 is not written to main memory (or L3). Up to four L3 caches can be arranged on a module. The L3 cache acts as a victim cache, storing cache lines that are evicted from L2; therefore, the L3 cache is not inclusive of the L2 cache.

5.2 Experimental Methodology

We used a 4-way POWER4 machine with two chips in which each chip is placed on a separate module. We use a variant of the SPECjbb2000 [29] benchmark for our experiments. SPECjbb2000 simulates a wholesale company whose execution consists of two stages. During startup, the main thread sets up and starts a number of warehouse threads. During steady state, the warehouse threads execute transactions against a database (represented as in-memory binary trees). The variant of SPECjbb2000 that we use is called pseudojbb: pseudojbb runs for a fixed number of transactions (120,000) instead of a fixed amount of time. We run pseudojbb with one warehouse thread on a single virtual processor. In our configuration, the pseudojbb run takes 25 seconds on a POWER4 workstation. We use an adaptive Jikes RVM configuration that has the adaptive optimization system (AOS) with a semispace garbage collector. The source code for Jikes RVM is from the September 26, 2003 head of the public CVS repository.

When run on a single virtual processor, Jikes RVM creates eight Java threads during execution in addition to the two Java threads created by pseudojbb: a *garbage collection* thread that executes only during garbage collection; a *finalizer* thread that finalizes dead objects and is executed infrequently; a *debugger* thread that never executes; and an *idle* thread that helps load balance Java threads when a virtual processor is idle, but because only a single virtual processor is used the idle thread never executes. As mentioned in Section 3 the VM also creates a *trace writer* thread to transfer trace records from a buffer to a trace file. The last three threads are related to the adaptive optimization system [6]. The *compilation* thread performs optimized compilations of methods selected by the *controller* thread, which processes online profile data recorded by the *organizer* thread.

5.3 Exploration

This section demonstrates two approaches to using the `Performance Explorer` to investigate performance phenomena. The first approach uses the `Performance Explorer` to look for general trends over time. We discovered the following two unexpected instruction per cycle (IPC) trends:

- Significant IPC improvement over time for an adaptive Jikes RVM configuration.
- Significant IPC degradation before GC.

Both trends were unknown before using the `Performance Explorer`. These trends over time would be difficult, if not impossible, to detect without a visualization tool.

The second approach uses the `Performance Explorer` to explore memory latency issues with respect to the POWER4 microarchitecture. Specifically, we used the `Performance Explorer` to explore the following questions:

- What is the impact on IPC and memory performance of context switches between different Java threads?[2]
- How do latency issues manifest in the POWER4 memory hierarchy?

Although the answers to these two questions might have been determined by performing a computation over the trace records directly, the `Performance Explorer` provides a unified approach to explore these questions.

5.4 General Performance Trends

The top graph in Figure 3 presents a graph of the IPC of a warehouse thread over time. For clarity of presentation, only the IPC trace records of the warehouse thread are displayed; all other Java threads are not shown. The noticeable gaps in the warehouse trace records are stop-the-world garbage collections, which disable Java thread scheduling when it runs.

Two general performance anomalies are noticeable in this graph. First, IPC improves over time. The IPC in the left corner is around 0.41, while the IPC in the right corner is above 0.49, a 20% increase. Second, before each of the GCs there is a significant IPC degradation or drop. We used the `Performance Explorer` to help us explore both of these phenomena.

5.4.1 Anomaly 1

To understand the IPC improvement over time, we used the `Performance Explorer` to chart each HPM event to determine the event or events that have a high correlation with the IPC. We found two such HPM events. The bottom graph in Figure 3 presents a warehouse thread's flushes per instructions completed over time.

A flush event may occur in a out-of-order processor for multiple reasons. A common reason is an out-of-order memory operation that executes and violates a data dependence. For example, the load of a memory location will be flushed if the load is speculatively executed before the execution of the store instruction to that memory location. When an instruction is flushed, all the instructions that are dispatched and occur in program order before the flushed instruction are also flushed. Hence, a flush event is expensive.

[2]In the past, a context switch, when control changes from one process to another, had been identified as having an impact on performance due to destroyed cache locality [25].

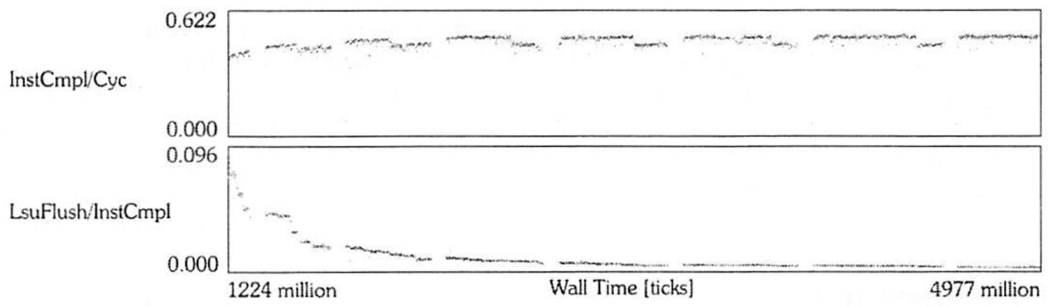


Figure 3: A graph over time of the warehouse thread's instruction per cycle and flushes per instructions completed.

Metric	JIT-base	AOS start (Δ%)	AOS end (Δ%)	JIT-opt0 (Δ%)
IPC	0.348	0.415 (+19.3%)	0.489 (+42.5%)	0.500 (+43.7%)
LSU_FLUSH/INST_CMPL	15.2%	7.0% (-54.2%)	0.3% (-97.4%)	0.1% (-99.3%)
INST_CMPL	4.18M	4.05M (+16.0%)	5.84M (+42.6%)	5.96M (+42.6%)

Table 2: Metrics across JIT and AOS configurations.

The POWER4 has multiple HPM events that count flush events. We have added together the values of the two dominant HPM flush events so that only one graph is displayed. The bottom graphs in Figure 3 illustrate that the flush rate changes dramatically over time. In the left corner, a flush event occurs for as many as 9.6% of the completed instructions while in the right corner a flush event occurs for as little as 0.4% of the completed instructions.

The Jikes RVM's adaptive optimization system (AOS) [6] compiles a method baseline (non-optimizing) compiler when it is first invoked. At thread switch time, the system records the method ID of the top method on the call stack and uses the number of times a method is sampled to predict how long a method will execute in the future. Using a cost/benefit model, the system determines when compiling a method at a particular optimization level has a benefit that outweighs the cost of the compilation for the expected future execution of the method. The system samples method IDs continuously throughout an application's execution, and optimizes methods whenever the model deems the optimizations are beneficial.

Table 2 provides insight into why the warehouse thread's performance improves over time. The first column specifies the two metrics that are graphed in Figure 3 and an additional metric of instructions completed (INST_CMPL). The table has four columns that contain metrics for the start and end of a run using the adaptive optimization system and two separate runs that use a non-adaptive strategy, which we call JIT configurations. In a JIT configuration, a method is compiled only once when it is first invoked. A JIT-base configuration uses the baseline compiler. A JIT-opt0 configuration uses the optimizing compiler at optimization level 0.[3] On a POWER4, the average execution time for SPECjbb2000 when one warehouse is run on 120,000 transactions is 27 seconds with JIT-opt0, and 175 seconds with JIT-base.

For this table, the metrics are computed as the average across warehouse trace records. The AOS start and end metrics were computed by taking the average of the first 14 and the last 14 warehouse trace records, respectively. The metrics for each of the JIT configurations is computed as the average over all of its warehouse trace records.

The first observation is that there is a large difference between the execution behavior of baseline compiled code (JIT-base) and the optimization level 0 compiled code (JIT-opt0). In particular, there is a 43.7% increase in IPC, a 99.3% decrease in flush events, and a 42.6% increase in instructions completed when going from baseline to optimization level 0. To understand this difference, we need to know how baseline compiled code differs from optimized code. The baseline compiler directly translates byte codes into machine code without any optimizations. In particular, no register allocation is performed and the Java expression stack is explicitly maintained with loads and stores accessing memory. Typically, after a value is pushed onto the stack it is popped off immediately and used. With baseline-compiled code, the number of flush events is high, 15.2% of the instructions completed. This is because the scheduling of out-of-order memory operations does not take into account the dependencies between memory locations: that is, the load instruction that models a pop to stack location $\mathcal{L}$ may be speculatively executed before the store instruction that models the push to stack location $\mathcal{L}$. In optimized code, the number of flush

[3]For the execution of pseudojbb that uses the adaptive optimization system, 612 methods are baseline compiled, and of those, 251 methods are recompiled: 125 methods are compiled at optimization level 0, 109 at level 1, and 17 at level 2. We show the metrics for a JIT-opt0 because the metrics at JIT-opt1 are similar, and although the metrics for JIT-opt2 degrade slightly, few methods are optimized at optimization level 2.

Metric	Total	Drop	Δ
INST_CMPL/CYC (IPC)	0.4924	0.4610	-6.4%
HV_CYC/CYC	0.0239	0.1249	+423.0%
EE_OFF/CYC	0.0197	0.0785	+300.0%
GRP_DISP_BLK_SB_CYC/CYC	0.0060	0.0258	+333.0%
LSU_SRQ_SYNC_CYC/CYC	0.0061	0.0170	+178.0%
STCX_FAIL/STCX_PASS+STCX_FAIL	0.0009	0.0040	+362.0%
LSU_LRQ_FULL_CYC/CYC	0.0008	0.0027	+250.0%

Table 3: Metrics that impact performance degradation before GC.

events is almost zero because values are loaded from memory into registers without explicitly modeling the Java expression stack.

Comparing the JIT configuration metrics, the impact of flush events on the number of completed instructions in a full 10 millisecond time slice is enormous:[4] 42.6% more instructions complete, when flushes are almost completely eliminated. In the AOS configuration, the execution behavior of the last fourteen warehouse trace records is very similar to the behavior of JIT-opt0. This is what is to be expected as the adaptive optimization system has had time to optimize the code that executes frequently. Running under the adaptive optimization system, when the warehouse thread starts executing all the warehouse methods are initially baseline compiled; however, because start up and steady state share code, some code that executes at the start of steady state is already optimized. As illustrated in Table 2, the AOS start has an expected behavior that falls between JIT-base and JIT-opt0 behavior.

5.4.2 Anomaly 2

To identify which HPM events correlate with the drop in IPC before each garbage collection, we used the `Performance Explorer` to select for each group of HPM events a representative set of warehouse trace records between two garbage collections and to compute the set's average for all HPM events (Total).[5] From this representative set, we used the `Performance Explorer` to pick the subset of trace records that represented the IPC drop (Drop) and computed its average for all HPM events. After these two computations, we identified the HPM events whose average values differed the most between Total and Drop. Table 3 presents our results. The first column identifies the HPM events. The HPM event values are normalized by dividing by cycles, except for STCX_FAIL, which is normalized by the number of STCX attempts. The second column presents the values for all trace records in the set (Total), the third column presents the values for the subset of trace records after IPC drops (Drop), and the final column presents how Drop's average changes as a percentage of Total's average. As can be seen, the IPC degraded by 6.4%, while there is a substantial increase (178–423%) in the percentage of the identified HPM events.

The set of events in Table 3 is eclectic. The HV_CYC (the processor is executing in hypervisor mode) and EE_OFF (the MSR EE bit is off) events specify the cycles spent in the kernel. The GRP_DISP_BLK_SB_CYC (dispatch is blocked by scoreboard), LSU_SRQ_SYNC_CYC (sync is in store request queue), and STCX_FAIL (a stcx instruction fails) events all have to do with synchronization. The LSU_LRQ_FULL_CYC event indicates that the load request queue is full and stalls dispatch.

Both the HV_CYC and EE_OFF events indicated increased kernel activity, and the difference in HV_CYC/CYC percentages accounts for just over 10% of all the cycles. The trace records for pseudojbb reflect both user and kernel mode execution; that is, the HPM counters continued counting when control enters the kernel. To determine if there was some underlying pathological Jikes RVM behavior that was causing increased kernel activity, we used the unix `truss` command to trace kernel calls. Other than `yield`, `_nsleep`, `thread_waitact`, and calls to the kernel HPM routines, there were no unusual kernel calls or increase in call activity that would explain the performance drop.

There is some indication that the IPC drop may be due to effective to real address translation (ERAT). The 128 entry IERAT and 128 entry DERAT are a cache for the 1024 entry TLB. We are investigating this hypothesis more thoroughly.

This anomaly illustrates the difficulty with determining performance anomalies with HPM information only. Deep microarchitectural, OS, and JVM knowledge is required to augment the HPM information.

5.5 Memory Wall Issues

We now use the `Performance Explorer` to explore how the scheduling of Java threads and garbage collection interact with the memory hierarchy. Our experiments use a semispace

[4]In a JIT configuration none of the AOS threads execute. In particular, there is no compilation thread; the cost of compiling a method is attributed to the application thread. Nevertheless, in a JIT configuration, the warehouse thread is interrupted only by the garbage collector and the infrequently executing finalizer thread. So once all the warehouse methods are compiled, every warehouse time slice runs for the full 10 millisecond quantum.

[5]Because the POWER4 has over two hundred events, but only 8 counters, this process had to be performed manually on many different trace files. In the future, this process could be incorporated into the `Performance Explorer` and automated.

garbage collector that divides the heap into two semispaces, *from* and *to*. The program allocates all objects in the *from* semispace. When garbage collector is triggered the collector copies all the live objects from the *from* space to the *to* space and switches the role of the *from* and *to* spaces. At the end of GC, all the live data is contiguous at the start of the newly-labeled `from` semispace.

Since Jikes RVM allocates code in the heap [3], the garbage collector moves around and compacts not just the data, but also the code that is not in the boot image (i.e., the compiler, standard libraries, etc.). The code is moved *through the data cache*. Thus after GC, references to instructions will miss in the instruction cache. The garbage collector does not move instructions in the boot image and thus after GC, references to these instructions may or may not miss (depending on how much of the cache is flushed by the garbage collector).

For the experiments reported in this section, we executed pseudojbb with the semispace garbage collector using the default adaptive heap size policy. The average amount of live data at the end of GC was around 25–30 MB. Before GC, the total heap space (excluding the *to* space) was about 100 MB.

5.5.1 Interaction of Threads with Memory System

We used the `Performance Explorer` to understand the impact of references and miss rates to the different memory hierarchy levels for both data and instructions of each Java thread. In particular, we compute the miss rate to L1 data cache by dividing the number of load misses from L1 (LD_MISS_L1) by the number of load references to L1 (LD_REF_L1). Unfortunately, the POWER4 HPM support does not provide miss and reference events to the other levels in the memory hierarchy, but does provides access counts: the number of times data is accessed from a particular cache. Therefore, we compute the number of references to a memory hierarchy level as the sum of the number of accesses to this level and to all lower levels. We compute the number of misses at a higher level in the memory hierarchy as the summation of the references at lower levels. The miss rate for a particular level is computed as the misses divided by the references to that level. For example, POWER4 HPM provides DATA_FROM_X events to specify the number of times data is accessed from the X level in the memory hierarchy. Thus, the L3 miss rate is (DATA_FROM_MM) / (DATA_FROM_L3 + DATA_FROM_MM).

Table 4 presents the references and misses metrics for the L1 instruction and data caches. The `Thread` column identifies the Java threads. The `Cycles` column identifies the percentage of total cycles spent executing this thread. The `Records` column reports the number of trace records captured for this thread. The `IPC` column reports instructions per cycle. The `Instructions` column reports the average number of instructions executed for each time quantum the Java thread is executed. The next two columns labeled `L1 Instructions` report the references and misses to the L1 instruction cache. The last two columns report similar values for the L1 data cache. The reference and miss rate computed metrics for the other levels of the memory hierarchy can be computed from Table 4 and the subsequent Figures 4 – 7

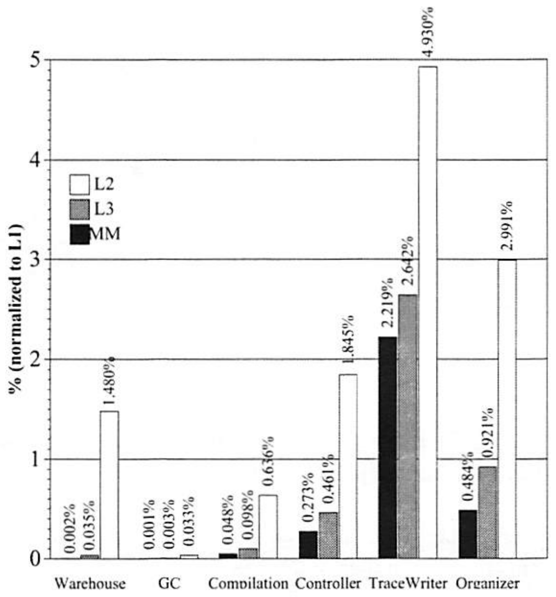


Figure 4: Instruction references to levels in the memory hierarchy normalized to all instruction references.

Figures 4 and 5 illustrate the references to L2, L3, and main memory as a percentage of all references for both instructions and data. The horizontal axis is the Java threads and the vertical axis is the percentage of all references for a particular thread. For each thread, there are three bars, one for each memory hierarchy level: `L2`, `L3`, and `MM`. Figure 4 shows that the three Java threads (warehouse, GC, and compilation), which consume 79% of all cycles, have a 98% hit rate for L1 instruction cache, indicating excellent instruction locality. The daemon threads, which execute infrequently and for a short interval of time, have worse L1 instruction locality. This is expected, because of their infrequent execution, little or none of their instructions will be in the cache, and because of there short execution duration, the cache misses have a short interval over which to be amortized. Of the three daemon Java threads, TraceWriter stands out as having the worst instruction locality with almost 10% of its instructions not found in the L1 cache: half of the 10% is found in the L2 instruction cache. TraceWriter is the least frequently executed daemon thread.

For data references, illustrated in Figure 5, the story is better and remarkable consistent across all threads. At most 5% of data is not found in the L1 cache for any of the Java threads. Both the warehouse and GC threads have the best locality with less than 3% of the data not found in the L1 cache. This implies that the working sets of the Java threads fit into the 64KB data cache. As is expected, the lower the memory level, the fewer the references: L2 has fewer than 3.4% of all references, and main memory has less than 0.2% of all references

Thread	Cycles	Records	IPC	Instructions	L1 Instructions		L1 Data	
					References	Misses	References	Misses
Warehouse	59.12%	1995	0.481	11,261,031,705	6,080,171,921	89,966,131	5,885,067,463	461,297,214
GC	13.96%	11	0.524	2,885,482,384	1,447,776,154	482,694	1,113,092,487	281,135,697
Compilation	6.46%	271	0.565	1,531,568,783	905,536,702	5,757,690	769,831,082	89,280,559
Controller	0.11%	141	0.212	11,867,348	8,730,297	161,096	8,496,326	835,170
TraceWriter	0.05%	82	0.097	2,411,860	1,395,686	68,802	1,599,610	447,691
Organizer	0.03%	141	0.166	2,037,304	1,322,770	39,568	1,307,913	145,696

Table 4: L1 data and instruction cache references and misses for different Java threads.

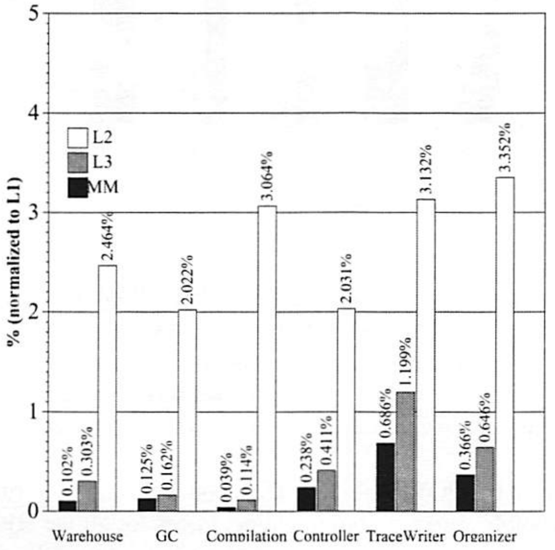


Figure 5: Data references to levels in the memory hierarchy normalized to all data references.

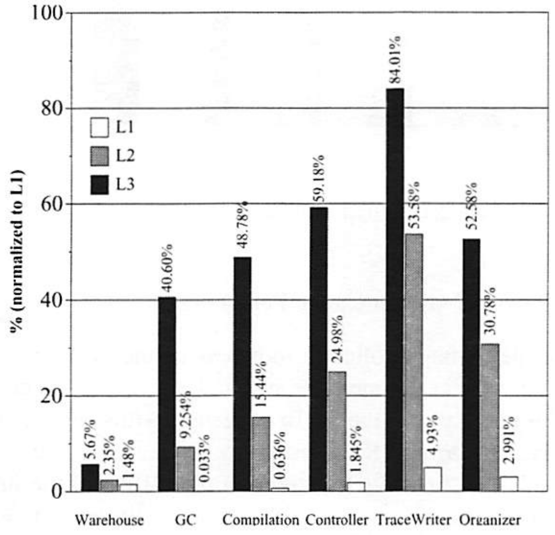


Figure 6: Instruction miss rates to levels in the memory hierarchy normalized to L1.

across all the Java threads.

Figures 6 and 7 illustrate the miss rates for the different levels of the memory hierarchy for both instructions and data. The figures illustrate that both the instruction and data L1 cache miss rates are small, less than 5% for instructions and less than 3.2% for data for all the Java threads. Furthermore, the figures illustrate that the L3 is of little help for all, but warehouse instructions: the L3 miss rate ranges from 33.57% to 84.01%. In general, the L2 miss rate is better for data than instructions. However, because the references to main memory and L3 are low, the high miss rates for L2 and L3 are not of a big concern.

Figure 6 shows that GC incurs a 0.033% instruction miss rate in L1. There are two reasons for this: (i) GC runs in uninterruptible mode where other Java threads cannot preempt its execution and evict its working set from the cache; and (ii) GC spends most of its time in a small inner loop which can easily fit in the cache. The instruction miss rates for the GC thread in L2 and L3 are not really relevant to performance since GC makes so few instruction references to L2 and L3.

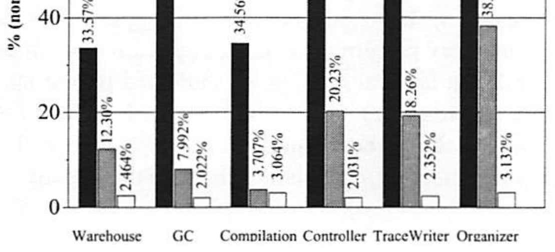


Figure 7: Data miss rates to levels in the memory hierarchy.

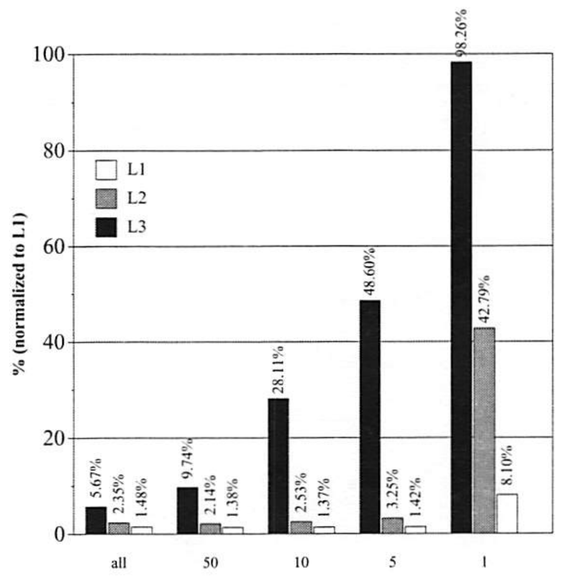


Figure 8: Instruction miss rates after a GC.

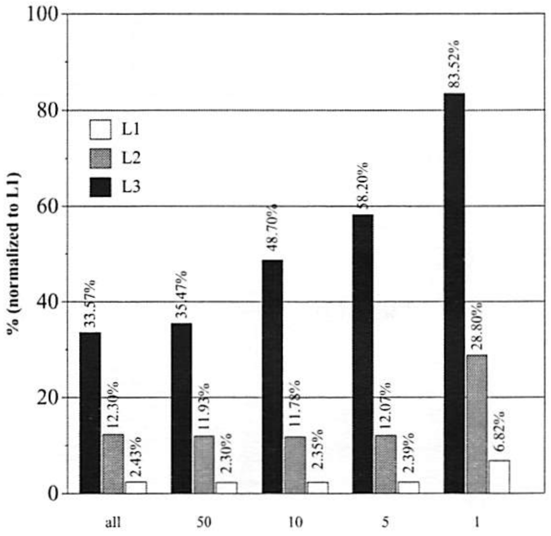


Figure 9: Data miss rates after a GC.

5.5.2 Effect of GC on Cache Performance

Because the garbage collector moves data and code, it can significantly affect the memory system behavior of the code that subsequently executes. To investigate this effect, we used `Performance Explorer` to extract miss rates for the first (1), the first five (5), the first ten (10), and the first fifty (50) warehouse trace records immediately after a GC. Figures 8 and 9 present this data for instructions and data, respectively. As a reference, the `All` bars present the overall cache miss rate of the warehouse thread.

From Figures 8 and 9 we see that both the instructions and data suffer increased cache misses immediately after a garbage collection. The misses in the L1 and L2 caches stabilize in less than 5 trace records. The much larger L3 cache takes even longer than 50 trace records to stabilize. The percentage of all references that are made to L2 is less than 0.30% for data and less than 0.05% for instructions for all but the first trace record after a GC. The percentage of all references that are to L3 is less than L2. The increased miss rates after a GC are also reflected in the IPC: the IPC in the first and first five trace records immediately following a GC are 0.069 and 0.357. In contrast the stable state IPC is 0.429.

These results clearly indicate that GC significantly degrades the memory performance of the application and it can take a significant amount of time, as indicated by the number of trace records, after a GC before the working set of the application is in the caches again. In future work we will investigate how other GC algorithms behave in this regard.

5.6 Discussion

We found that having a tool to manipulate the HPM data is indispensable. With over sixty thousand events in one 30-second execution of pseudojbb, some kind of tool is required to organize the HPM data. Surprisingly we found that, in many cases, visualization alone was not sufficient to detect trends. Alternatives, such as selecting subsets of trace records and computing average metric values over the subset, were required.

Because the POWER4 has only 8 counters, multiple runs of pseudojbb were required to collect traces for all the HPM events. The `Performance Explorer` is currently designed to work on one trace file at a time. Therefore using the `Performance Explorer` to understand trends across all the HPM events is rather tedious as a new window has to be opened for each trace. Extending the functionality of the `Performance Explorer` to perform more complex computations and support for better report generation would be helpful.

The performance degradation before GC demonstrates why traces for all HPM events are required as it is difficult to always know a priori what subset of HPM events are needed.

It took us a while to determine the correct computed metrics to explore the performance issues related to memory latency. One difficulty was understanding how HPM events can be combined to compute metrics. In general, the HPM data are part of the puzzle and getting the complete picture may require additional information or experiments. We found that the HPM data can help to determine what additional information is required. For example, we used JIT configurations to understand the IPC improvement over time for the adaptive configuration of Jikes RVM.

6 Related Work

This section surveys related work on software tools that collect, analyze, and visualize hardware performance counter data and other performance-related work for Java.

6.1 Accessing Hardware Counters

Several library packages provide access to hardware performance counter information, including the HPM toolkit [15], PAPI [11], PCL [9], and OProfile [26]. These libraries provide facilities to instrument programs, record hardware counter data, and analyze the results. We extend the functionality of existing libraries to obtain hardware performance data in a virtual machine. Specifically, we extend Jikes RVM to collect thread-specific, temporally fine-grained hardware counter data in an SMP environment.

The Digital Continuous Profiling Infrastructure provides a powerful set of tools to analyze and collect hardware performance counter data on Alpha processors [5]. The system includes tools to collect accurate profile data with very low overhead and to analyze the profile data using many performance metrics. The system works on unmodified executables on multiprocessors and collects time-based hardware counter samples of program counter values. VTune [32] and SpeedShop [35] are similar tools from Intel and SGI, respectively. Our work differs in that we are interested in correlating the hardware counter data to high-level program constructs, such as Java threads, to distinguish the effects from the VM and user applications, in an SMP environment in a temporal manner.

Ammons et al. [4] correlate hardware performance counter information to frequently executed program paths. They use flow- and context-sensitive data-flow analysis techniques to collect hardware counter data along program paths instead of just individual statements or procedures. Although this provides fine-grained information, the overhead of recording hardware counter data along the paths increases runtime by an average of 70%. The overhead of collecting and storing our HPM trace files is less than 2% in our VM.

IBM's Performance Inspector [30] is a collection of profiling tools. It includes a patch to the Linux kernel providing a trace facility and hooks to record scheduling, interrupt and other kernel events. The package contains tools for sampling-based profiling (any performance counter can be used to trigger sampling), manual instrumentation-based profiling (measuring per-thread time/performance counts), and exception-based basic block profiling. Our work profiles several performance counters at once, does not require instrumentation by hand, and is tightly integrated with our VM, resulting in access to information about the runtime system (like compilation and garbage collection).

6.2 Profiling Java Workloads

The Java Virtual Machine Profiler Interface (JVMPI) defines a general purpose mechanism to obtain profile data from a Java VM [31]. JVMPI supports CPU time profiling (using statistical sampling or code instrumentation) for threads and methods, heap profiling, and monitor contention profiling. Our work differs in that we are interested in infrastructure that measures the architectural level performance of Java applications.

Java middleware and server applications are an important class of emerging workloads. Existing research uses simulation and/or hardware performance counters to characterize these workloads. Cain et al. [13] evaluate the performance of a Java implementation of the TPC-W benchmark and compare the results to SPECweb99 and SPECjbb2000. The TPC-W benchmark models an online bookstore, and the Java implementation is a combination of Java Servlets that communicate with a database system using JDBC. Cain et al. use hardware counters to measure the performance of the entire benchmark on an IBM multiprocessor with eight RS64-III processors. They also use simulation to experiment with new architectural features. Our infrastructure enables us to distinguish the performance between the various threads of the VM and application, on different processors, at regular time intervals.

Luo and John [21] evaluate SPECjbb2000 and VolanoMark on a Pentium III processor using the Intel hardware performance counters. Seshadri, John, and Mericas [28] use hardware performance counters to characterize the performance of SPECjbb2000 and VolanoMark running on two PowerPC architectures. The focus of both studies was to compare Java server applications to the SPECint2000 benchmarks, which are written in C. They obtain aggregate counter information over a significant portion of the benchmarks running a single processor, whereas we gather hardware performance data on multiple processors on a per thread basis at regular intervals.

Karlsson et al. [20] characterize the memory performance of Java server applications using real hardware and a simulator. They measure the performance of SPECjbb2000 and ECPerf on a 16-processor Sun Enterprise 6000 server. Karlsson et al. use the hardware counters to measure coarse-grained events. The results are for the execution of the entire benchmark, and they do not distinguish between the VM and the application. Our infrastructure enables us to obtain fine-grained performance measurements information in real time.

Dufour et al. [16] introduce a set of architecture- and virtual machine-independent Java bytecode-level metrics for describing the dynamic characteristics of Java applications. Their metrics give an objective measure of aspects like array or pointer intensiveness, degree of polymorphism, allocation density, degree of concurrency, and synchronization. Our work analyzes workload characteristics on the architectural level, covers both application, library and the virtual machine behavior, and investigates behavioral patterns over time.

6.3 Statistical Performance Analysis

Recent work uses statistical techniques to analyze performance counter data. Eeckhout et al. [17] analyze the hardware performance of Java programs. They use principal component analysis to reduce the dimensionality of the data from 34 performance counters to 4 principal components. Then they use hierarchical clustering to group workloads with similar behaviors. They gather only aggregate performance counts, and they divide all performance counter values by the number of clock cycles. Ahn and Vetter [1] hand-instrument several code regions in a set of applications. They gather data from 23 performance counters for three benchmarks on two different parallel machines with 16 and 68 nodes. Then they analyze that data using different clustering algorithms and factor analysis, focusing on parallelism and load balancing. We visualize behavior over time, require no instrumentation, and allow the analysis of any kind of derived metric.

6.4 Performance Visualization

Mellor-Crummey et al. [23] present HPCView, a performance visualization tool together with a toolkit to gather hardware performance counter traces. They use sampling to attribute performance events to instructions, and then hierarchically aggregate the counts, following the loop nesting structure of the program. Their focus is on attributing performance counts to source code areas, whereas our focus is attributing them to processors and threads. They provide only metrics aggregated over the complete runtime. We show the value of metrics over time, which is important for understanding the application behavior in a virtual machine with a rich runtime system.

Miller et al. [24] present Paradyn, a performance measurement infrastructure for parallel and distributed programs. Paradyn uses dynamic instrumentation to count events or to time fragments of code. It can add or remove instrumentations on request, reducing the profiling overhead. Metrics in Paradyn correspond to everything that can be counted or timed through instrumentations. The original Paradyn does not support multithreading, but Xu et al. [34] introduce extensions to Paradyn to support the instrumentation of multithreaded applications. Our infrastructure contains full support for gathering hardware performance counters and is tightly integrated with the Java virtual machine's thread scheduler, which allows us to gather accurate performance measures for the complete system with very low overhead.

Zaki et al. [36] introduce an infrastructure to gather traces of message-passing programs running on parallel distributed systems. They describe Jumpshot, a trace visualization tool, which is capable of displaying traces of programs running on a large number of processors for a long time. They visualize different (possibly nested) program states, and communication activity between processes running on different nodes. The newer version by Wu et al. [33] is also capable of correctly tracing multithreaded programs. We focus on tracing a single process on one SMP computer. Instead of tracing communication activity and user-defined program states of MPI (Message Passing Interface) programs, we gather and visualize the hardware performance of Java applications on a virtual machine.

Pablo, introduced by Reed et al. [27], is another performance analysis infrastructure focusing on parallel distributed systems. It supports interactive source code instrumentation, provides data reduction through adaptively switching to aggregation when tracing becomes too expensive, and introduces the idea of clustering for trace data reduction. DeRose et al. [14] describe SvPablo (Source View Pablo), loosely based on the Pablo infrastructure, which supports both, interactive and automatic software instrumentation and hardware performance counters, to gather aggregate performance data. They visualize this data for C and Fortran programs by attributing the metric values to specific source code lines. We focus on low overhead, fully automatic tracing of temporal data for a Java virtual machine and application. Our visualizations provide detailed information about the hardware performance and the behavior of the virtual machine.

7 Future Work

This paper uses the `Performance Explorer` to explore the performance behavior of one pseudojbb warehouse thread on one virtual processors. We are interested in performing additional experiments with more warehouses on one or more virtual processors, as well as exploring other benchmarks, to determine how the POWER4 HPM data may help to understand application behavior.

To provide temporal data, the system currently uses the context-switching among Java threads as the delimiter for counting intervals. Because the stop-the-world garbage collector prevents thread switching while it is executing, the system views a garbage collection, which can be longer than 10ms, as a single trace record. Because it is desirable to provide finer granularity for VM operations that disable context-switching, we plan to explore the addition of a separate trigger for capturing HPM information that will be in effect even when thread switching is disabled.

The adaptive optimization system of Jikes RVM's uses a cost/benefit model to determine which methods of an application should be optimized and at what optimization level [6]. We can leverage this system to attempt to selectively capture critical computations of the code in an automatic manner. Following in the direction of Arnold et al. [7], we could capture detailed profile information for only those methods that execute often. Specifically, we can instruct the optimizing compiler to automatically insert VM calls to partition the most time consuming computations and capture HPM information for these computations.

In this paper we explore using interactive visualization to drive performance analysis, which enables one to see large- and small-scale patterns, and to look at the data from any desirable perspective. A complimentary approach is to use statistical analysis to reduce the dimensional space. This is a worthwhile approach that we plan to explore.

8 Conclusions

We describe a system for exploring the low-level behavior of Java applications. Our system generates traces containing data obtained from hardware performance monitors and provides an interactive graphical user interface to explore the data. Although prior work presents tools for accessing hardware performance monitors, our work is unique in that it correctly attributes behavior to Java threads in a multithreaded system running on an multiprocessor. Our work is implemented in the context of Jikes RVM.

We demonstrate the usefulness of our tools by applying them to understanding the behavior of pseudojbb, a variant of the SPECjbb2000 benchmark. We demonstrate that our tools are able to identify as-yet unknown performance characteristics of the benchmark and are able to guide us in understanding the reasons for the observed performance characteristics. This understanding is essential for designing new high-payoff optimizations and for tuning applications and run-time systems for the best performance.

9 Acknowledgments

The authors thank Laureen Treacy for proofreading earlier drafts of this work. Amer Diwan and Matthias Hauswirth were partially supported by NSF grants CCR-0085792 and CCR-0133457. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16. IEEE Computer Society Press, 2002.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. *ACM SIGPLAN Notices*, 34(10):314–324, October 1999. Published as part of the proceedings of OOPSLA'99.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, May 1997. Published as part of the proceedings of PLDI'97.

[5] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Sun tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[6] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

[7] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, November 2002. Proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'02).

[8] Steve Behling, Ron Bell, Peter Farrell, Holger Holthoff, Frank O'Connel, and Will Weir. *The POWER4 Processor Introduction and Tuning Guide*. Redbooks. IBM Corporation, International Technical Support Organization, 2001.

[9] Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr. PCL - the performance counter library. http://www.fz-juelich.de/zam/PCL.

[10] Stephen Blackburn, Perry Cheng, and Kathryn McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *26th International Conference on Software Engineering*, May 2004.

[11] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, November 2000.

[12] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[13] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Nuevo Leone, Mexico, January 2001.

[14] Luiz DeRose and Daniel A. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.

[15] Luiz A. DeRose. The hardware performance monitor toolkit. In Rizos Sakellariou, John Keane, John Gurd, and Len Freeman, editors, *Proceedings of the 7th International Euro-Par Conference*, number 2150 in Lecture Notes in Computer Science, pages 122–131, Manchester, UK, August 2001. Springer-Verlag.

[16] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, 2003.

[17] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th Annual ACM*

SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 169–186, 2003.

[18] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264. IEEE Computer Society, 2003.

[19] Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

[20] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 217–228, Anaheim, CA, February 2003.

[21] Yue Luo and Lizy Kurian John. Workload characterization of multithreaded Java servers. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 128–136, Tucson, AZ, November 2001.

[22] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[23] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. HPCView: A tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, October 2001.

[24] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[25] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Santa Clara, CA)*, pages 75–84, 1991.

[26] Oprofile. http://oprofile.sourceforge.net, 2003.

[27] Daniel A. Reed, Ruth. A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1993.

[28] Pattabi Seshadri, Lizy John, and Alex Mericas. Workload characterization of Java server applications on two PowerPC processors. In *Proceedings of the Third Annual Austin Center for Advanced Studies Conference*, Austin, TX, February 2002.

[29] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[30] Bob Urquhart, Enio Pineda, Scott Jones, Frank Levine, Ron Cadima, Jimmy DeWitt, Theresa Halloran, and Aakash Parekh. Performance inspector. http://www-124.ibm.com/developerworks/oss/pi, 2004.

[31] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, February 2000.

[32] Intel VTune performance analyzers. http://www.intel.com/software/products/vtune.

[33] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[34] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.

[35] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, November 1996.

[36] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.

vBlades: Optimized Paravirtualization for the Itanium Processor Family

Daniel J. Magenheimer and Thomas W. Christian
Hewlett-Packard Laboratories

dan.magenheimer@hp.com, twc@fc.hp.com

Abstract

Virtualization of an "uncooperative" architecture often has severe performance consequences. Paravirtualization has recently been suggested as a solution to performance issues, but it introduces unacceptable supportability problems. The HP Labs vBlades project has identified a novel hybrid approach – which we call optimized paravirtualization. We examine methods for both virtualizing and paravirtualizing the Itanium processor, and then demonstrate optimized paravirtualization to maximize performance while simultaneously minimizing supportability concerns.

1 Introduction

As computer performance increases, it becomes more desirable to utilize available performance flexibly and efficiently. On even the smallest personal computer, multiprocessing enables several applications to share the processor. Other techniques such as virtual memory and I/O device abstraction support the illusion that each application controls all physical resources, or even more resources than are physically available. In the pursuit of efficiency, one thing has remained constant: general-purpose operating systems assume that they have complete control of the system's physical resources. The operating system thus assumes responsibility for allocation of physical resources, communication and management of external storage.

Virtualization changes that. Similar to the way that a general-purpose operating system presents the appearance to multiple applications that each has unrestricted access to a set of computing resources, a virtual machine manages a machine's physical resources and presents them to one or more operating systems, creating for each the illusion that it has full access to the physical resources that have been made visible.

Virtual machines were the subject of extensive research in the 1960s and 1970s [1, 2, 3, 4, 5]. Originally developed to enable expensive mainframe resources to be shared by several operating systems or other privileged applications, they were quickly applied to other problem domains including system management, software development and security [6, 7, 8]. Increasingly, data centers are demanding rapid adaptability, requiring a single server to run one operating system for a period of time then be quickly redeployed to run another operating system serving a different purpose. Some high-end servers today provide hardware-based partitioning mechanisms [9] to allow multiple operating systems to share the same server. On an even broader scale, the grid promises the capability of sharing underutilized, geographically dispersed computing resources [10]. The resource management capability that results from virtual machines can help solve these problems by separating the operating system from the underlying hardware in ways that can yield new levels of flexibility.

Researchers have devoted years to the study and deployment of virtual machines for the x86 (IA-32) platform. As a result, much work has appeared in the literature describing the issues that arise in virtualizing the x86 architecture [11, 12]. The Itanium (IA-64) processor was introduced in 1999, beginning a family of 64-bit processors intended for high-end servers and workstations. Co-developed by Intel and HP, Itanium is known for the high performance made possible by its explicitly parallel architecture, but Itanium has another attribute that has been less widely publicized: it was expressly designed with features that provide increased security for computer systems [13]. These features make Itanium eminently suitable for future Adaptive Enterprise and grid applications. It is useful to understand the virtualization issues for this architecture and determine how the benefits of virtualization will apply. We explore these issues and describe how we have made use of virtualization on Itanium for the HP Labs vBlades virtual server project.

2 vBlades Approach and Overview

A Virtual Machine Monitor (VMM) is a software layer that virtualizes the available resources of a computer and multiplexes them among one or more *guest* operating systems. Implementing a VMM can be fairly straightforward if the target architecture was designed to support virtualization but quite complex if not. The Instruction Set Architecture (ISA) of a machine must conform to certain constraints for it to be *fully virtualizable* – that is, able to be represented as an exact duplicate by the VMM [4]. Unfortunately, these constraints are not met for the predominant x86 architecture, nor are they met for Itanium.

Ideally, an operating system should be able to run without modification on a VMM, while retaining the illusion that it is running directly on physical hardware and owns all resources. Different methods have been suggested to support this illusion on an architecture that is not fully virtualizable; such methods almost always result in significant performance degradation.

Some VMMs intentionally compromise the virtual machine interface in exchange for greater performance. For example, VMware provides an add-on driver which, when loaded by a Windows guest, greatly reduces the I/O overhead [14]. Other VMMs provide an explicit API and allow or require a guest operating system to port to the VMM, a technique the Denali project [15, 16, 17] has named *paravirtualization*.

The Xen [18] team demonstrated how paravirtualization improves performance, scalability and simplicity at the cost of a small set of changes to the guest operating system. Xen has crystallized a set of design principles that we paraphrase here:

1. Existing application binaries must run unmodified.
2. Multiple commercially available operating systems must be supported.
3. Paravirtualization is necessary for performance and security, especially on "uncooperative" machine architectures.
4. Hiding the effects of resource virtualization is generally unnecessary and impacts not only performance and security but also correctness.

These design principles explain the justification for paravirtualization but they say nothing about its major disadvantage: operating system modifications, especially significant ones, can be problematic in the real world.

First, if substantial modification is required, the operating system provider may summarily reject the necessary changes. This is true not only for proprietary operating systems but also for open source operating systems. For example, the simple changes required for Xen's XenoLinux impact architecture-independent code in the Linux distribution. Historically, there has been some reluctance to change this code for architecture-specific features.

Second, in a research or academic environment, operating system variations are common and it is probably reasonable to expect a separate operating system image for operation in a virtual environment. In a production environment, loading a different operating system image is unwieldy. For a commercial operating system provider, doubling the number of distributed operating system images is a supportability issue and almost certainly unacceptable.

To address these concerns, we suggest two additional design principles for the "Xen of Virtualization":

5. Operating system changes for paravirtualization must be minimized and limited to architecture-dependent code.
6. One paravirtualized operating system image must be capable of running either native or as a guest under the VMM.

The HP Labs vBlades project is exploring virtualization on Itanium to support a virtual server environment. The vBlades goals include:

- Concurrent execution of multiple operating system images, each with their own application set, in isolated protection domains with security and privacy enforced by hardware.
- Optimal server utilization through allocation and dynamic management of virtual servers that map to fractional, integral or aggregated physical servers.
- Comprehensive measurement, monitoring and control capabilities for detailed performance analysis, QoS monitoring, resource management and accounting.
- Resource management and security protocols that enable integration of vBlades virtual servers into utility data centers and the grid.

VBlades supports both virtualization and paravirtualization. The vBlades *hypervisor* handles emulation of privileged operations while the vBlades *virtualization abstraction layer* (VAL) provides the API used by ported guests. The components may be used separately or together. That is, operating systems may run fully virtualized, undertake a complete port to the VAL or use the facilities in combination. By starting with a fully virtualized system, making performance measurements for selected benchmarks then adding VAL calls to resolve performance issues, an optimal balance can be found between the magnitude of the required modifications and performance. We call this hybrid approach *optimized paravirtualization*.

3 Virtualizing the Itanium Processor

As was previously noted, the present Itanium architecture is not fully virtualizable [4]. This section describes some of the most important issues with Itanium virtualization and the approaches used by vBlades to resolve the issues. It is intended to be illustrative, not comprehensive.

3.1 CPU Virtualization

3.1.1 Ring Compression

Four privilege levels or *rings* are supported on Itanium. Privilege level zero (PL0) is the most privileged and the only level at which privileged instructions may be executed. Itanium operating systems typically utilize only two privilege levels: the operating system runs at PL0 with all privileges and user processes run unprivileged, usually at PL3. PL1 and PL2 are generally unutilized.

VBlades takes advantage of the unused levels by employing the traditional VMM *ring compression* technique. VBlades demotes a guest to privilege level two (PL2), reserving both PL0 and PL1 for its own operation. All unprivileged instructions, whether executed by the guest or one of the guest's processes, execute normally and at full performance. Privileged instructions executed by the guest result in the delivery of a privileged operation fault, which is fielded by the vBlades hypervisor.

One difficulty Itanium has with ring compression is that a guest can easily determine the privilege level at which it is executing, a problem commonly known as *privilege leakage*. Several Itanium non-privileged instructions allow the Current Privilege Level (CPL) to be examined. A guest concerned about potential security vulnerabilities might refuse to boot or run if it determines that it is running virtualized. A similar difficulty arises if a guest makes use of all four privilege levels. Both of these issues can be avoided, but only with significant performance impact and/or by utilizing sophisticated instruction transformation techniques. Fortunately, these issues rarely arise in commercially available operating systems.

3.1.2 Emulation of Privileged Operations

When a privileged operation fault results from a guest attempt to execute a privileged operation, the vBlades hypervisor decodes and emulates the instruction. Rather than faithfully emulate the precise semantics of the instruction, vBlades usually will choose to apply its own interpretation to virtualize the effects of the instruction. For example, a guest may utilize Itanium's `rsm psr.i` instruction to turn off delivery of interrupts. VBlades does not actually disable interrupts but instead just records the guest's intent and honors the fact that any interrupts intended for that guest should not be delivered until further notice.

A complication may arise in the process of emulating an Itanium privileged instruction. Some architectures provide a special register – often called the Instruction Register (IR) – to record the currently executing instruction. Itanium does not provide an IR so the vBlades hypervisor must utilize other state information to read the instruction from memory. However, all current Itanium implementations support independent translation buffers for instruction and data access. Since the original fetch occurred as an *instruction* access and the second read is a *data* access, the hypervisor must be prepared to sustain a data translation fault. If this occurs, the hypervisor must search the translation tables to find the correct translation for the instruction.

3.1.3 Exceptions / Interrupts

Itanium defines a set of conditions that result in exceptions and interrupts (collectively referred to as interruptions) and also defines a privileged Interruption Vector Address (IVA) register that defines the base of a code table. Different types of interruptions are delivered to different places in the IVA-based code table. Certain state bits are disabled automatically on delivery of an interruption. For example, interrupt delivery and interrupt state collection are both turned off.

All of this virtualizes in a relatively straightforward way: The vBlades hypervisor records the guest's IVA register and, for interruptions that need to be handled by the guest, it adjusts state appropriately and delivers control to the guest at the guest's interruption handler.

One complication arises in certain situations involving the Itanium *register stack engine* (RSE). The register stack enables automatic register renaming in order to accelerate handling of procedure call data, while the RSE handles memory traffic between the register stack and backing store memory. The RSE operates concurrently with the processor and may attempt to load or store data that results in a virtual addressing fault. The normal Itanium interruption delivery mechanism is used for these faults but a special bit is set in the processor state to indicate that the fault resulted from an RSE memory operation. Simultaneously, another processor status bit is cleared to disable RSE activity.

The complication occurs because the latter bit – the RSE Current Frame Load Enable (RSE.CFLE) bit – is not architecturally visible and cannot easily be modified. According to the Itanium specification, this bit is enabled only – and unconditionally – on execution of any procedure return (`br.ret`) or return-from-interruption (`rfi`). In a native operating system, the OS interruption handler simply resolves the fault prior to returning control to the faulting process. However, in many cases the vBlades hypervisor must cede control to the guest to resolve the fault. When this happens, RSE activity is automatically enabled, resulting in immediate recurrence of the fault.

Several approaches were investigated to resolve this rare but tricky problem. On the first design attempt, the register stack was forced into a known stable state prior to delivery of control to the guest for any interruption using the Itanium `cover` instruction. However, certain guest interruption handlers were unable in some non-RSE fault cases to deal with a "pre-covered" register stack. Next, we attempted to track the other RSE fault indication bit (ISR.ri) to deliver the stack "pre-covered" only when an RSE fault had occurred. Tracking this state proved to be problematic. Finally, we settled on a delayed approach that we call *lazy cover*. We allow the fault to recur upon delivery to the guest and, when it does, special code recognizes the recurrence. We then cover the register stack and redeliver the fault. This results in an extra vBlades-to-guest interruption delivery but the situation happens so rarely that performance is not an issue.

3.1.4 Privilege-sensitive Instructions

Privilege leakage is one example of a visible difference that occurs as a result of guest privilege demotion. Itanium has several other instructions that have privilege-related issues:

- The previously mentioned `cover` instruction has a side effect that saves important register stack information in a privileged register. However, the side effect only occurs under certain circumstances that are restricted to PL0 execution.
- `thash` and `ttag` are unprivileged instructions that surface information from privileged virtual memory data structures.
- A bit in the processor status register – PSR.sp – controls whether the performance data registers can be read by non-privileged instructions. However, if unprivileged access is denied, attempted reads do not trap but instead simply return zero.

These instructions, which behave differently depending on current privilege level, can be referred to as *privilege-sensitive instructions.*

A common VMM technique for dealing with privilege-sensitive instructions involves dynamic transformation of the instruction stream. Because of the bundling of Itanium's explicitly parallel instructions, further constrained by functional unit asymmetry and bundle templates that limit the types of instructions the bundle may contain, dynamic transformation on Itanium can be difficult [19]. The vBlades design is capable of incorporating a dynamic transformation mechanism but static instruction replacement has proven sufficient for vBlades purposes. We avoid complicated replacement choices by directly replacing each privilege-sensitive instruction with a similar privileged instruction.

The `cover` instruction has a single encoding with no variations and can be replaced with a `break.b` instruction. But `thash` and `ttag`, which each have two register arguments, are more complicated and require a brief discussion of register usage on Itanium.

Nearly all Itanium instructions that access registers utilize a seven-bit register field, allowing usage of Itanium's 32 64-bit general-purpose registers and the 96 additional automatically renumbered registers on the register stack. These register stack registers, numbered 32 to 127, are heavily used by the procedure calling mechanism and normally contain procedure parameters and local variables. At procedure entry, an Itanium `alloc` instruction specifies the portion of the register stack that is used by this procedure, starting at register number 32. For example, a procedure may indicate that only registers 32 through 40 will be used, in which case registers 41 through 128 will not be available in the current register stack. Interestingly, Itanium specifies that while *writes* to numbered registers currently unavailable in the register stack result in an illegal operation trap, *reads* from those registers simply return a zero – without resulting in an illegal operation trap.

VBlades takes advantage of this last point. While user-level code may use registers numbered in the sixties or higher, in system code such register usage is rare and in low-level system code it is exceedingly rare. VBlades steals the high 64 register numbers of the source register for two privileged instructions and uses these for the privileged instruction replacements for `thash` and `ttag` as shown in Figure 1. This static translation precludes the possibility of a guest using a register numbered higher than 63 for any of these four instructions, but that has yet not proven to be a problem.

```
thash rx=ry → tpa rx=r(y+64), 0≤y<64
ttag rx=ry → tak rx=r(y+64), 0≤y<64
```

Figure 1 – Modified thash and ttag Instructions

3.2 Memory Virtualization

Studies have shown [20, 21] that memory loads and stores make up a large percentage of an instruction stream. Consequently, a machine's virtual memory architecture is designed to ensure that virtual memory accesses proceed efficiently and securely. To maximize performance, vBlades must stay out of the way of the vast majority of the memory accesses of a guest and its user processes, while retaining the capability to intercede if a guest exceeds its bounds, maliciously or otherwise.

3.2.1 Address Spaces

As with most modern architectures, Itanium provides the capability to isolate the address space of different processes. To do so, it provides eight privileged *region registers* that participate in each virtual address translation. The range of values that can be contained in a region register is implementation-dependent and must be obtained through a call to the Itanium-architected Processor Abstraction Layer (PAL) firmware, which returns the number of bits in the region register. Setting a region register to a value outside of this range results in a fault.

VBlades intercepts the guest's PAL call and always returns the architectural minimum, thus limiting each guest to 2^{18} address spaces. Since setting a region register is a privileged operation, vBlades can intercede to reserve some values for its own purposes and partition the set of address spaces among the guests, securely restricting the virtual addressing capabilities of each guest.

3.2.2 Metaphysical Memory

In some situations, an operating system may choose to override the protections afforded by the machine's virtual addressing mechanism in order to directly access real machine memory. Itanium controls whether accesses are virtual or physical with bits in the privileged Processor Status Register (PSR). Once in physical mode, an Itanium native operating system can access any memory address, read or write device control or data registers or, by accessing a non-existent physical address, cause a machine check and crash the system.

In order to enforce security, vBlades cannot allow a guest to access physical memory directly. To prevent this, vBlades inserts an extra layer of indirection between a virtual address and its corresponding physical address. Although the concept of an intermediate layer is not unusual in VMM implementation, nomenclature is confusing and not standardized; to clearly differentiate it from real machine physical memory, we refer to this layer as *metaphysical addressing*[1]. VBlades intercepts attempts by the guest to transition from virtual mode to physical mode and instead places the guest in metaphysical mode by adjusting region registers so that virtual addresses translate to a reserved per-guest address space.

Once in this mode, the guest believes that it is directly accessing physical memory but the physical addresses it is using are actually virtual addresses that vBlades controls and monitors. When a guest access to a metaphysical address results in a virtual addressing fault, vBlades first validates the address to ensure isolation, and then resolves the fault invisibly to the domain by providing the appropriate mapping. Note that since this mechanism utilizes all of the machine's translation hardware, performance is preserved for guests that frequently access physical memory.

Rather than use an extra level of addressing indirection, some VMMs simply partition physical memory among the guests. This limits either the number of guests or the amount of physical memory assigned to each. A valuable side effect of the vBlades approach is that it can utilize the indirection to provide additional features. Just as a native operating system utilizes virtual memory and disk paging to create the illusion for each of its processes that more memory exists than is actually available, vBlades can oversubscribe physical memory for its guests. It can demand load or swap out lightly utilized memory, share read-only memory segments between similar guests and adjust access to physical memory as needed to maintain a specified quality-of-service level.

3.3 Timer Virtualization

A native Itanium operating system marks the passing of time through the use of a free-running Interval Time Counter (ITC) and an Interval Time Match (ITM) register. The period of the ITC is obtained through a call to the PAL firmware. The operating system triggers

[1] Merriam-Webster defines metaphysical as: "of or relating to…a reality beyond what is perceptible to the senses." Since metaphysical memory represents physical memory in a way that is not perceptible to a guest, we believe this usage is appropriate, though admittedly light-hearted.

timer interruptions by setting a value in the privileged ITM register. When the value of the ITC matches the value in the ITM, an interrupt is generated. On Itanium, firmware may take control of the machine for an indefinite period of time, during which interrupt delivery is disabled and the operating system is effectively sleeping. An Itanium operating system must be resilient to such blank periods. When the operating system finally sees the interrupt, the value in the ITC may greatly exceed the value in the ITM – perhaps by as much as one or more quanta. The timer interrupt service routine must be capable of recognizing this situation and recovering appropriately.

VBlades takes advantage of this to avoid virtualization of time. A guest may be out of context for an extended period while other guests or vBlades are running and must be capable of recovering from this situation. However, even if a guest recovers it is not clear what the impact will be on its processes, for example, when accounting for resource usage. We have considered a software interrupt to notify a guest that it has been sleeping, but have not yet implemented it or seen a requirement for it. It remains to be seen if this will be required to serve the needs of some guests or if a virtual time mechanism (such as the one proposed by Xen) will need to be architected and implemented.

4 Paravirtualizing Itanium

As others have observed, paravirtualization can serve a number of objectives. In Denali, an abstract interface different from the underlying x86 hardware is convenient for supporting thousands of underutilized virtual machines. For Xen, knowledge of the underlying API allows more efficient access of x86 page tables while isolating potentially malicious guests. Since other purely virtual mechanisms could suffice, we posit that every use of paravirtualization is a way to improve performance.

Paravirtualization of Itanium is no different. The first vBlades design required a complete guest port based on the assumption that any virtualization would result in unacceptable performance degradation. All privileged operations required a VAL call and no privileged operation trapping was supported. As measurement and monitoring capabilities were added, we were able to quantify the frequency of privileged requests. We found that the vast majority of VAL calls were due to interrupt enable/disable requests, TLB miss processing and system calls. In a second tier were calls for timer handling, external interrupt handling and context switches. This led us to focus tuning efforts on improving the highest frequency operations.

4.1 The Privileged State Communication Block (PSCB)

On every Itanium interruption, certain privileged registers provide information to assist the operating system in resolving and recovering from the interruption. For example, on all interruptions the last value of the instruction pointer and the processor status register are preserved so that execution can be resumed (with an `rfi` instruction), if appropriate, when interruption processing is complete. Some other examples: On a TLB miss, the faulting address is provided; on a "break" fault (commonly used for system service calls) the instruction contains an immediate value that is provided to the interruption handler.

When a native operating system processes an interruption, several of these privileged registers are read and/or written and each register access requires execution of a privileged instruction. To avoid this, vBlades defines the Privileged State Communication Block (PSCB), a shared-memory area used to record the information contained in these privileged registers and enable communication of the information to and from guest interruption handlers.

In many cases, the PSCB contains an exact match of the privileged register that would be seen by a native operating system. For example, the Interruption Status Register (ISR) is delivered unchanged. In other cases, the register is "virtually" identical; that is, it has been adjusted by vBlades according to virtualization constraints. An example of this is the "current privilege level" bit in the virtual interrupt processor status register (IPSR) which is set at interrupt delivery to zero to reduce privilege leakage.

4.2 Some Serialization Required

Because Itanium is an explicitly parallel architecture, some processor state modification instructions require a non-privileged serialization (`srlz.i` or `srlz.d`) instruction to be executed to ensure the effects of the state modification take place before a subsequent instruction that depends on those effects. For example, writes to the previously mentioned Itanium ITM register may not result in a timer interrupt until a `srlz.i` instruction is executed. For certain PSCB fields, and under certain circumstances, the vBlades VAL requires a similar mechanism.

For example, the "interrupt delivery enabled" field is the virtual equivalent of the hardware `psr.i` bit. If a guest wishes to disable interrupts, it clears this field and interrupts are pended – noted but not delivered to the guest – until further notice. If the guest wishes to enable interrupts, it sets the field to a non-zero value. However, vBlades only checks this for *subsequent* interrupts; if any interrupts are pending at the time the guest enables interrupts, delivery is delayed unless the guest invokes a VAL synchronization service call, as shown in Figure 2. In order to expedite this check, another PSCB field specifies whether any interrupts are pending. If interrupt arrival frequency is substantially lower than interrupt disable/enable frequency, this model can substantially reduce the need for VAL calls.

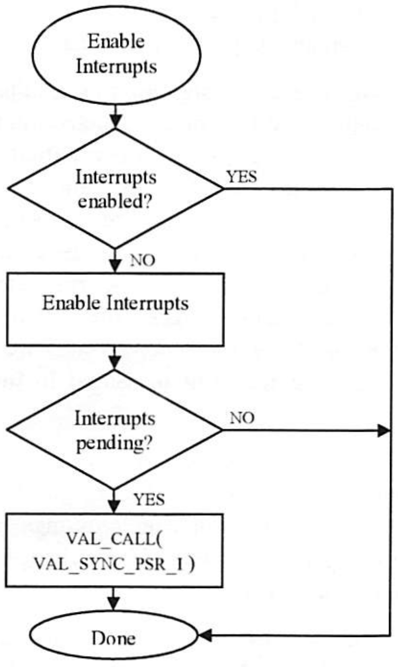


Figure 2 – Enabling Interrupts with Paravirtualization

4.3 Batching

In many cases, replacing emulation of a single privileged operation with a single VAL service call provides negligible savings. However, if a group of privileged operations can be replaced by a single VAL service call, significant performance improvements can result. For example, when a guest is performing a task switch it will usually update several (or all) of the region registers with address space values appropriate for the new task. Rather than making a VAL call for each individual region register, one VAL service allows all eight to be updated with a single call.

4.4 Transparent Paravirtualization

The performance advantages of paravirtualization are evident. As previously noted, there are disadvantages to requiring a separate binary for running native vs. running as a guest on a virtual machine. If an operating system can determine whether or not it is running virtualized, it can make optimal execution choices at runtime and the same binary can be used. We call this *transparent paravirtualization*.

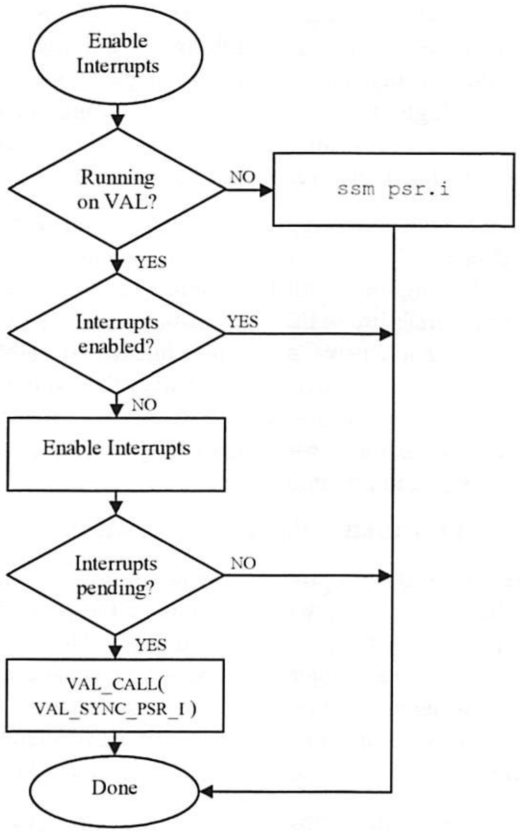


Figure 3 – Enabling Interrupts using Transparent Paravirtualization

VBlades utilizes a reserved bit in a privileged configuration register to let the operating system know whether or not it is running virtualized. According to the Itanium architecture definition, reserved bits in the configuration register are always set to zero. When the vBlades hypervisor executes the privileged instruction that returns this register, it sets one of the reserved bits to one. Thus, an operating system can execute this instruction early in the startup process and conditionally set a global variable to record whether or not it is running as a vBlades guest. Once this variable is set, subsequent transparent code can test the variable and react accordingly as illustrated in Figure 3.

In a transparently paravirtualized operating system, this conditional test may occur with relatively high frequency; indeed, every piece of paravirtualized code requires the test. When running as a guest, the incremental cost of the additional test is small relative to virtualization overhead. We conjectured that the cost when running native would also be small. First, in a fully paravirtualized guest, the number of tests is at most one per privileged instruction. Second, the frequency of privileged instructions in all but the most system-centric micro-benchmarks is at least two to three orders of magnitude lower than unprivileged instructions. Third, a well-defined paravirtualization interface eliminates many privileged instructions. Finally, high frequency access to the conditional test variable ensures its presence in cache memory, guaranteeing a low cycle count for the conditional test.

To test our conjecture, we ran a simple but non-trivial benchmark: Linux compiling itself. The difference was indeed negligible, with the magnitude dwarfed by the natural variability in the benchmark results; we expect a more comprehensive set of benchmarks to show that degradation is less than 0.1%. If true, this would show that the performance impact of transparent virtualization on a native operating system is, as its name would imply, transparent.

4.5 Optimized Paravirtualization

One of our design principles requires limiting changes to the guest, yet we wish to minimize the performance degradation of the paravirtualized guest. This is clearly an iterative and subjective process: Some guests may have stringent requirements on code change, while others may be much more focused on performance. We refer to the process as *optimized paravirtualization*.

To measure the degree of change to the guest, we define the set of changes necessary to implement paravirtualization as the *porting footprint*. Changes to the guest fall into two categories: *invasive* changes and *supporting* changes. Invasive changes are those that affect one or more existing source or build files. Supporting changes are newly added source or build files that provide VAL support code necessary for interfacing to the vBlades VAL but do not affect existing code; these are generally linked in as a library. We believe that invasive changes have, by far, the most significant impact on operating system maintenance. Consequently we restrict our definition of porting footprint to include only invasive changes.

To support data-driven performance decisions, vBlades is highly instrumented. It records and tabulates all VAL calls, privileged operations, exception deliveries, etc. This level of detail is not only crucial for porting but can also provide an interesting perspective on the operation of the original pre-ported guest.

The vast majority of application and guest instructions executed in any benchmark are unprivileged, execute at full speed and are thus irrelevant to a comparison. Since the guest is executing unprivileged, all privileged instructions must either be emulated by the vBlades hypervisor or replaced and paravirtualized through VAL calls. We will refer to these collectively as ring crossings.[2] Obviously, each ring crossing is slower than the native privileged instruction it replaces – perhaps by two to three orders of magnitude. Consequently, reducing the total number of ring crossings improves performance. Further, a VAL call is somewhat less costly than hypervisor emulation since the hypervisor must fetch and decode the privileged instruction. Thus, replacing an emulated privileged instruction with an equivalent VAL call also improves performance.

With this in mind, we present ring-crossing results from the previously introduced benchmark (Linux compiling itself) at different stages of optimized paravirtualization of Linux 2.4.20. Prior to the execution of the benchmark, all vBlades counters are zeroed; thus privileged instructions and VAL calls necessary to initialize the system are ignored. The ring crossing results of the different stages are graphically represented in Figure 4. On the second y-axis we show the cumulative porting footprint measured in lines of code.

In stage 0, only a minimal set of changes is introduced into Linux to allow it to run as a vBlades guest. There are approximately 474 million ring crossings, all of them due to privileged instructions. These changes have a porting footprint of 46 lines.

In stage 1, we replace Linux interrupt enable/disable code with the VAL call mechanism described in Section 4.2. Because of the highly organized nature of the Linux source code, the vast majority of code that enables or disables interrupts uses preprocessor macros defined in a single include file; these macros utilize the Itanium `rsm` and `ssm` instructions. We redefine these macros using a patch that has a porting footprint of only four lines. With this minor change, almost 111 million (23%) of the privileged operations are eliminated and replaced with less than one million VAL calls, reducing ring crossings to 363 million.

In stage 2, we introduce a vBlades-specific Interruption Vector Table (IVT). In Itanium, the IVT is the entry point for all interruption handlers, including synchronous exceptions such as TLB faults as well as timer and external device interrupts. Since Itanium

[2] Technically, there are at least two ring crossings for each hypervisor or PAL call but we omit this detail for the purpose of clarity. Only the units of measurement are affected, not the impact on performance.

interruption handlers obtain and manipulate state by reading and writing privileged registers, the IVT contains many privileged instructions. As previously described, these can be replaced with normal loads and stores to the PSCB.

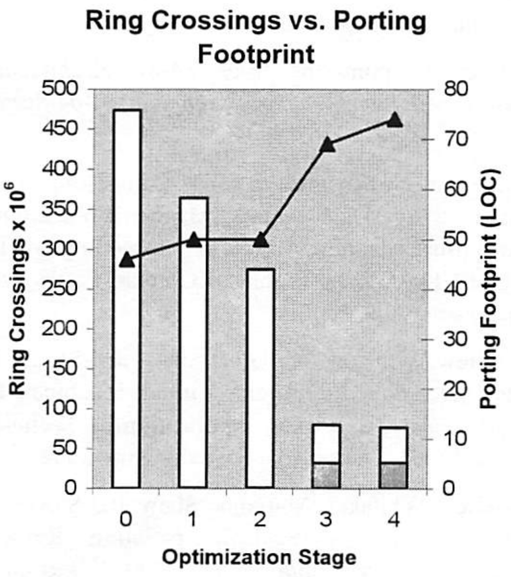


Figure 4 – Ring Crossings vs. Porting Footprint

Linux running on Itanium must indicate the location of the IVT by storing the address in privileged `cr.iva` register exactly once early in architecture dependent startup code, prior to the possibility of any interruption. Replacing the original Linux IVT with a VAL-aware IVT could be as simple as conditionally assigning a different location to `cr.iva`. However, the VAL sensing code also must execute prior to any interruption. So instead we allow the original code to set `cr.iva` to point to the original Linux IVT, then reset it in the VAL sensing code to point to the VAL-aware IVT. As a result, there is no additional porting footprint for this change. The resultant reduction in ring crossings, however, is significant – now down to 274 million.

Every entry into the Linux kernel must have a corresponding exit, and just as the IVT reads numerous privileged registers, many of these same privileged registers must be written when returning to interrupted user code. In stage 3, we replace the central Linux kernel exit code with a VAL-aware version, a change that requires a porting footprint of 19 lines and see a dramatic improvement in the number of privileged operations, which has been reduced to 48 million. We also see the first significant increase in VAL calls – a total of 32 million, visible on the bar chart as the crosshatched portion of the bar. One VAL_RESUME call, the equivalent of the Itanium `rfi` instruction, is made for each kernel exit. The total number of ring crossings is now 80 million.

In stage 4, we examine the benefit of the region register updates seen previously as an example of batching. When performing a task switch, Linux/ia64 changes five region registers using five consecutive privileged instructions. We replace all five privileged instructions with a single VAL call, using a patch that has a porting footprint of five lines. The benefits of this stage, though significant, are not as remarkable as the previous stages. We have replaced about 3.8 million privileged operations with about 0.7 million VAL calls, a net reduction of over 3 million ring crossings.

In some cases, a large reduction in ring crossings that yields significant performance improvements can be obtained with a very small porting footprint. In other cases, changes with a larger porting footprint may result in a negligible performance change. Through careful experimentation and measurement a suitable balance can be achieved.

We redirected the vBlades project prior to completion of extensive application benchmarking and without ports of guests other than Linux. In an earlier prototype, a small suite of benchmarks was used to compare performance of Linux running fully paravirtualized against native Linux. While this prototype made simplifying assumptions regarding I/O, the observed performance degradation was approximately 1-2%, comparable to the paravirtualized x86 measurements published by the Xen team.

5 Conclusions

We have described virtualization and paravirtualization issues for the Itanium processor family. Combining these techniques using optimized paravirtualization allows a balance to be reached between maximizing performance and minimizing the porting footprint (and maintenance impact) for the guest operating system; we believe that, with a small porting footprint, performance can approach native operation. Finally, we have introduced transparent paravirtualization, which enables a single operating system image to run either on a native system or a VMM, improving maintainability at essentially no cost.

6 Acknowledgments

The other members of the vBlades team are: Rob Gardner, Chris Hyser, Bret McKee, Christopher Worley and Mike Wray. This work evolved from the HP Labs Secure Platform Architecture project that also included

Bill Worley and John Worley. Christophe de Dinechin, Todd Kjos, Jonathan Ross and Jean-Marc Chevrot suggested some Itanium virtualization techniques. David Mosberger and Stéphane Eranian's text [22] provides an excellent overview of Itanium, Linux and the port of Linux to Itanium; it was exceptionally useful in the Linux/IA-64 port to vBlades.

7 References

[1] Robert P. Goldberg, "Architecture of Virtual Machines," *AFIPS Conference Proceedings*, 1973 NCC, AFIPS Press, Montvale, NJ.

[2] Carl J. Young, "Extended Architecture and Hypervisor Performance," *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, MA, 1973.

[3] Robert P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, pp. 34-45, June, 1974.

[4] Gerald J. Popek and Robert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, 17(7), pp. 412-421, July 1974.

[5] Gerald J. Popek and Charles S. Kline, "A Verifiable Protection System," Proceedings of the International Conference on Reliable Software, pp. 294-304, Los Angeles, CA, 1975.

[6] Thomas C. Bressoud and Fred B. Schneider, "Hypervisor-based Fault Tolerance," ACM Transactions on Computer Systems, 14(1), pp. 80-107, 1996.

[7] Gerald J. Popek and Charles S. Kline, "A Verifiable Protection System," Proceedings of the International Conference on Reliable Software, pp. 294-304, Los Angeles, CA, May, 1975.

[8] James E. Smith, "An Overview of Virtual Machine Architectures," http://www.ece.wisc.edu/~jes/902/papers/intro.pdf, October 2001.

[9] Hewlett-Packard Company, "HP Integrity Superdome Technical White Paper", available from http://www.hp.com/, 2003.

[10] R. Figueiredo, P. Dinda and J. Fortes, "A Case for Grid Computing on Virtual Machines", *Proceedings of the 23rd International Conference on Distributed Computing Systems* (ICDCS 2003), May 2003.

[11] John S. Robin and Cynthia E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", *Proceedings of the Ninth USENIX Security Symposium*, August 2000.

[12] Carl A. Waldspurger, "Memory Resource Management in VMware ESX Server," *Proceedings of the 5th Symposum on Operating System Design and Implementation* (OSDI 2002), December 2002.

[13] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture*, 2000.

[14] Ganesh Venkitachalam and Beng-Hong Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," Proceedings of the 2001 USENIX Annual Technical Conference, Boston, Massachusetts, June 2001.

[15] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," Technical Report 02-02-01, University of Washington, 2002.

[16] Andrew Whitaker, Marianne Shaw and Steven D. Gribble, "Denali: A Scalable Isolation Kernel," *Proceedings of the Tenth ACM SIGOPS European Workshop*, St. Emilion, France, 2002.

[17] Andrew Whitaker, Marianne Shaw and Steven D. Gribble, "Scale and Performance in the Denali Isolation Kernel," Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI), 2002.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebarger, Ian Pratt and Andrew Warfield, "Xen and the Art of Virtualization," *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[19] Christophe de Dinechin, Todd Kjos, Jonathan Ross and Jean-Marc Chevrot, Hewlett-Packard Company, internal communication.

[20] Joseph A. Lukes, "HP Precision Architecture Performance Analysis," *Hewlett-Packard Journal*, vol. 37, pp. 30-39, August 1986.

[21] John Hennessy, Norman Jouppi, Forrest Baskett, Thomas Gross and John Gill, "Hardware/Software Tradeoffs for Increased Performance," *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Palo Alto, CA. March, 1982.

[22] David Mosberger and Stéphane Eranian, *IA-64 Linux Kernel: Design and Implementation*, Copyright 2002, Prentice Hall Professional Technical Reference.

Kernel Plugins: When A VM Is Too Much

Ivan Ganev, Greg Eisenhauer, Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{ganev,eisen,schwan}@cc.gatech.edu

Abstract

This paper presents *kernel plugins*, a framework for dynamic kernel specialization inspired by ideas borrowed from virtualization research. Plugins can be created and updated inexpensively on-the-fly and they can execute arbitrary user-supplied functions such that neither safety nor performance are compromised. Three key techniques are used to implement kernel plugins: (1) hardware fault isolation, (2) dynamic code generation, and (3) dynamic linking. Hardware fault isolation protects kernel-level services from plugin misbehavior, dynamic code generation enables rapid online creation of arbitrary plugins, and dynamic linking governs the kernel/plugin interface.

We discuss the design and implementation of the kernel plugin facility, as well as its advantages and shortcomings. Its use is demonstrated by a range of micro- and macro-benchmarks and a real-life application featuring plugins that dynamically transcode images served by a high-performance kernel web server. Benefits realized from plugins can be both qualitative (adapting services to clients' needs), and quantitative (improving performance through co-location of application plugin code with kernel services). Plugins are implemented in GNU/Linux on the Intel x86 platform. Reported performance results include plugin upcalls in 0.45-0.62 μS, dynamic code generation in 4 mS, and linking/unlinking in 3.1/1.6 μS for an image grayscaling plugin – a dynamically code generated 66-line function written in a subset of C. All results are measured on an 866 MHz Pentium III.

1 Introduction

Recent years have seen explosive growth in ubiquitously available computing power and network bandwidth, and we have witnessed the advent of novel products like smart mobile phones, wireless PDAs, and tablet PCs. These advances have spurred a wide range of applications, including Internet radio stations, peer-to-peer networks, and cellphone-based photography. Common to all such ubiquitous devices and applications is the need to guarantee high quality of service despite unpredictable availability of platform resources and dynamically varying user needs. Two methods of addressing these issues have traditionally been resource reservation and system adaptation. Because of its ability to provide firm guarantees, the former has enjoyed strong popularity in real-time and mission-critical applications. Such firm guarantees, however, come at the cost of markedly lower resource utilization and that fact has made adaptation the method of choice for non-critical and consumer applications [2, 20].

The need for adaptation has spurred an extensive body of research into dynamically extensible systems at all levels of the computing and networking infrastructure, from library-based middleware techniques [6, 10], to extensible operating systems [4, 12, 15, 22, 27], to programmable network processors [13], and even active networks [24].

Techniques for implementing runtime extensions must balance *performance* with *safety* concerns. Efforts to achieve higher performance can degrade the safety and security of services, while efforts to bolster security may negatively impact performance by requiring time- and resource-consuming runtime checks. Consequently, a wide variety of solutions for extending kernel-level services have been proposed, ranging from approaches based on 'little languages' [16], to entirely new operating system kernels [4, 12]. One solution is to place extensions inside a virtual machine (VM) [25], completely isolating them from the rest of the system and thereby avoiding the need to trust them. The simplicity and safety of this approach is accompanied by some drawbacks, however, including: (1) the performance of virtual machines is inferior to that of native hardware [22], and (2) multiple OSes running in multiple VMs can complicate resource sharing and result in inefficient resource usage.

Our research seeks a middle ground between the complete isolation offered by virtual machines and the unsafe practice of system extension by adding new kernel modules. Our approach combines the use of virtualization techniques with dynamic binary code generation and dynamic linking, resulting in the *kernel plugin* framework for runtime kernel extension.

A kernel plugin is made up of one or more application-supplied program functions that extend some kernel-level service. It is installed upon a client application's request and runs on its behalf. Plugins are designed to cooperate with, rather than replace, kernel-level services. Their interactions are controlled, so that a plugin only has privileges explicitly granted to it by the kernel. A well-defined plugin/kernel interface governs all such interactions. The efficient plugin mechanism permits rapid creation, update, and removal of plugins, thereby encouraging applications to frequently avail themselves of the mechanism's advantages.

Plugins are realized for the standard Linux kernel and the popular x86 hardware platform, offering a safe, efficient service extension mechanism to a broad set of developers. Our implementation achieves both high performance and safety by integrating three key techniques: (1) hardware fault isolation, (2) dynamic code generation, and (3) dynamic linking. Hardware fault isolation protects kernel services from misbehaving plugins. Dynamic code generation enables rapid runtime creation of custom plugins. Dynamic linking governs the kernel/plugin interface.

A key result of our research is the high performance of plugins, made possible by using isolation techniques borrowed from virtualization research [8, 9, 25], and by promoting frequent system adaptation through efficient plugin creation and deletion. For instance, plugin invocation costs are 0.45-0.62 μS on an 866 MHz Pentium III depending upon the number of plugin parameters. In addition, plugin creation and setup costs are low, thereby encouraging their use in ways that are not easily implemented with coarser-grain mechanisms. Code generating a sample 66-line C code plugin on the same platform takes 4 mS, while linking and unlinking take 3.1 μS and 1.6 μS, respectively.

In the remainder of this paper, we describe the design and implementation of kernel plugins on Intel x86 platforms running the GNU/Linux operating system. Kernel plugins are evaluated with micro- and macro-benchmarks, as well as with a realistic application – an accelerated web server augmented by a plugin specializing the data it delivers to clients. This example, evaluated in detail, is on-the-fly transcoding of image data, streamed from the server's disk to its communication link.

2 Related Work

While safe runtime kernel extension has previously been addressed in the literature, unfortunately such functionality is not generally available in commonly used operating systems. Several classes of solution techniques have been proposed:

Programming Language Techniques
In the SPIN operating system, the safety of kernel extensions is based on the properties of the Modula-3 type-safe programming language and a trusted compiler [4]. Furthermore, because SPIN's kernel extensions use relatively heavyweight external compile/link/execute facilities, creation costs must be amortized over extended and frequent use. As a result, SPIN extensions are best suited to long-lived functionality.

The Open Kernel Environment (OKE) [5] employs a variation of the same idea, substituting the type-safe Modula-3 with Cyclone, an 'elastic' customizable version of C, and trust management integrated with the compiler.

In contrast to these schemes, kernel plugins are designed to be lightweight, agile, and easy to adapt on-the-fly. Plugin creation, invocation, and removal overheads are very low and do not involve execution of external compilers or linkers. Furthermore, our facility implements both preemption and isolation and thus does not need to trust any binaries outside the kernel.

Proof-Carrying Code
Proof-carrying code [18] is a mechanism for safety verification of code that requires that a 'safety proof' is attached to each piece of code, certifying its adherence to a pre-defined 'safety policy'. The proof is such that quick validation is possible without cryptography or external references. Despite those desirable properties there are three drawbacks to proof-carrying code.

The first and foremost one is that generating a comprehensive safety policy for non-trivial code is very hard. The difficulty results from the fact that the policy needs to cover all obvious and implied rules and invariants of the execution environment. Furthermore, there is no way to guarantee the completeness of the policy itself. Second, the method has scaling issues because the safety proof's size grows large rather quickly. As an example, a trivial function summing two numbers under a basic safety policy is quoted to have 60 bytes of code and 430 bytes of safety proof [18]. Finally, no automatic proof generators exist.

Kernel plugins provide an alternative – an engineering solution that achieves native code performance and safety without the burden of a proof or type-safe language restriction.

Software Fault Isolation
SFI approaches [26] rely on rewriting the machine code of extensions so that memory accesses and jump targets are checked and instrumented, thereby restricting them to the scope of the extension's protection domain. Only after such *sandboxing* is an extension allowed to execute. Program interpretation is a related approach in

which extensions are executed by a trusted interpreter that enforces safety.

Typical examples of such extensible kernels are VINO [21], which relies on SFI, and packet filters like the Berkeley Packet Filter [16], which implements an interpreted 'little language' for custom, in-kernel, packet filtering rules. The primary problem with these approaches is that the price of safety is non-trivial performance degradation, which makes them less appealing for high-performance applications. The performance of type-safe language extensions is quoted to be 10% to 150% worse than regular C code, and SFI can be as much as 220% slower [8]. In comparison, kernel plugins do not incur per-instruction execution overheads. Plugin code generation is a one-time cost, significantly smaller than compilation alternatives and amortized over the lifetime of the plugin.

Hardware Fault Isolation

HFI relies on hardware-provided memory management features to enforce the isolation between the kernel and extensions. This is the same method that traditional operating systems use to isolate their kernels from user-space applications. It also forms the basis for most 'virtualization' and 'isolation' systems, which can be viewed as very coarse-grain extension mechanisms. Notable examples include the VMware [25] and Virtual PC [9] virtual machines, as well as the library operating systems supported by Exokernel [12], the Denali isolation kernel [28], and Xen [3] – a new VM monitor that defines an abstract VM to which kernels are then ported, reportedly achieving close to native performance.

Palladium [8] also uses hardware features to achieve extension isolation, but on a somewhat finer grain and without striving to provide a complete virtualization environment. It limits its scope only to untrusted kernel modules, and uses segmentation and privilege-checking hardware to ensure that they cannot interfere with the kernel proper. While Palladium's strategy results in better performance compared to virtual machines, it still restricts system adaptation to relatively coarse-grain kernel modules, and limits the dynamic use of such extensions because it requires off-line module compilation.

Kernel Plugins

Like some of the above approaches, we choose to employ a hardware-based scheme, exploiting the x86 architecture's segmentation hardware and unused privilege rings to provide isolation. Specifically, the x86 hardware provides 4 'privilege ring levels'. Typical operating systems use ring-0 (most privileged) and ring-3 (least privileged) for kernel and user modes, respectively. Kernel plugins utilize one of the unused privilege rings. Thus, memory protection and control-flow restrictions are enforced entirely in hardware, causing no discernible performance degradation. This is a popular isolation approach employed by all x86 virtual machine projects of which we are aware, as well as the implementation of intra-address space protection in Palladium.

Unlike VMware and VirtualPC style VMs, however, we do not strive to provide the illusion of a dedicated machine. Instead, we define a streamlined, lightweight execution environment in a manner which is more meaningful and fitting to a plugin's purpose of customizing existing services rather than deploying new ones. Unlike Exokernel, Denali, and Xen, we do not modify host architectural assumptions and require no porting or reimplementation of host-kernel subsystems that do not need to be extensible. Finally, unlike Palladium we strive to achieve finer granularity and enable runtime online adaptation while keeping setup overheads low. Experimental results presented in this paper demonstrate that kernel plugins experience no additional runtime costs per instruction. We also show that the overhead of protected control transfers to and from plugins are both small and predictable.

3 Motivation

Previous work [4, 8, 12, 20, 21, 22] has already demonstrated that application-specific extension of operating system kernels can be a key contributor to attaining high end-to-end performance. A wide range of specializations exist that can easily be realized using plugins – the spectrum of opportunities spans virtually all subsystems of a modern OS kernel. Plugins could augment a file system with custom caching or prefetching algorithms, or modify a TCP stack's back-off strategy to reflect loss properties of a particular client's link. They could enhance core system services like scheduling, by providing scheduling hints in the guise of payoff functions, or extend memory management by specializing the behavior of page replacement algorithms. Finally, kernel plugins can even be useful in high-performance kernel servers like the Linux accelerated web servers TUX and kHTTPd. Some examples of the rich set of application-specific plugins that can be deployed are (1) dynamic compression and decompression of data to effect trade-offs in server vs. client CPU needs and/or required transmission bandwidth, (2) runtime downsampling techniques reflecting a clients' preferences for fidelity vs. timeliness, (3) region-of-interest type transformations, removing unnecessary data from a communication stream, etc.

The following example kernel plugin usage scenarios have guided our research:

Smart Filtering

One usage of plugins is to permit end users to directly affect data production, transmission, and reception at the kernel level. For instance, if certain data is not of current interest to the recipient, it can be eliminated early in the receiving OS kernel, rather than being transferred to user level only to be discarded. Similarly, if only subsets of data are of interest to specific recipients, then source-based and client-specific data filtering may be implemented with plugins [10]. Alternatively, plugins can be used for 'valuation' of information being captured, processed, transmitted, or received, by applying payoff or utility functions to it. Research has shown that such utility functions can be a very useful adaptation tool.

Intelligent Introspection
Another possible domain of use for kernel plugins is system monitoring and instrumentation [23]. The idea is to deploy code that is tailor-made for its specific purpose and to allow it to evolve dynamically with the needs of the client, instead of having to measure and export a large and generic set of metrics. For instance, an NFS client experiencing degradation of service can dynamically instrument its server's disk and network subsystems to discover where the bottleneck is and adapt or possibly work around it.

Runtime Adaptation
A final example of a kernel plugin usage scenario is to enable low-overhead dynamic self-adaptation of a system's behavior, perhaps as a response to changes in monitored conditions. For instance, the NFS client from our previous example determines that there is a disk head scheduling bottleneck and adapts by pushing into the NFS server an aggressive prefetch algorithm customized to its current access patterns.

4 Design

4.1 Approach

The success of any OS facility is strongly linked to its performance characteristics and ease of use. Thus, a principal goal of our framework is to provide an effective, efficient, and easy to use extension mechanism. The following properties guided our design:

- *Generality:* The API should be generic and avoid targeting a specific kernel service.
- *Functionality:* Unnecessary restrictions should be avoided on what constitutes valid plugin code. Plugin creation, use, and deletion should be possible in runtime, using both statically pre-compiled and dynamically generated code.
- *Safety:* The core kernel should be protected from direct or unintended manipulation by plugin code.
- *Efficiency:* Implementation overheads should be less than or comparable to alternatives.

Kernel plugins attain these properties by combining three key technologies: (1) hardware fault isolation, (2) dynamic code generation, and (3) lightweight dynamic linking.

Hardware fault isolation protects the core kernel from the untrusted plugins and helps to avoid costly per-instruction runtime overheads. It provides an engineering solution to the isolation problem without the complexity and overheads inherent in programming-language techniques, proof-carrying code, or software-fault isolation.

While a library of pre-compiled adaptation strategies that clients can choose from can go a long way, sometimes applications need tailor-made solutions. Adapting file system prefetching to irregular access patterns, or filtering out or digesting parts of complex objects to transfer are but a few such examples.

Dynamic code generation, thus, serves a two-fold purpose. First, it provides a common language for arbitrary and cross-platform runtime adaptation in a heterogeneous environment, and second, it promotes performance by translating extensions into native machine code able to run at full speed on bare hardware.

It is important to realize that we do not mean to discount the usefulness of libraries of pre-compiled plugins. Such libraries are certainly instrumental for complex, static codes like fast Fourier transforms, JPEG encoding/decoding, etc. Rather, we propose to augment such libraries with a complementary mechanism that is able to adapt to variable runtime conditions.

Dynamic linking controls the kernel/plugin interface. It enhances the plugins' expressive power by permitting collaborative compositions of plugin functions to perform complex tasks.

4.2 Plugin Runtime

The base plugin mechanism is a simple abstraction of an 'execution environment'. This environment, termed the *plugin runtime*, registers, handles, and manipulates the kernel plugins of a single extensible entity. It can be thought of as defining a streamlined abstraction of a tiny virtual machine. As our aim is not to emulate a particular systems platform but to create a clean and efficient extension environment, we are able to design for simplicity and reap the benefits of efficiency.

Each runtime has a restricted, but well-defined API providing the means to add new plugin functions, as well as to execute and delete existing ones. Multiple runtimes may exist simultaneously at any given time, each managing the extension functionality of a single client or client instance. Each instance of an extensible entity creates its own runtime and dynamically populates it with client-supplied plugins. The resulting multiplicity of runtimes serves a threefold purpose. It allows extensibility on a per-instance basis, prevents plugin namespace pollution, and isolates related or cooperating plugin functions within a single runtime. Actual coordination and cooperation of plugin functions of a single client is left to the client itself, with the runtime only providing the glue primitives to enable it.

As an illustration, consider two separate kernel services: a kernel http daemon (as in our sample application), and a kernel NFS server. Each server instance creates a runtime for its plugins and populates it upon its clients' request. The http daemon's threads might install any number of image manipulation plugins, whereas the NFS daemon's threads might install various data compression algorithms. The separate runtimes ensure that the namespaces of unrelated plugins belonging to different clients are disjoint and that unrelated data and symbols cannot be named or invoked.

Built-in Plugins
Sometimes application-specific plugin code will need to call on certain kernel functions to achieve its goals, e.g. enqueuing a packet or reading/writing a block from/to disk. To accommodate such *callbacks* seamlessly within our framework they are represented as a kind of plugin. These 'built-in' plugins are explicitly added to the runtime by the kernel service it extends. Even though they act as kernel callbacks, within the restricted plugin environment they are indistinguishable from a regular 'dynamic' or user-supplied plugin. That is to say that they are invoked and used in exactly the same fashion. Immediately after its creation, a runtime's namespace contains only a default set of available built-ins listed in Figure 1. They perform basic namespace maintenance expected from the dynamic linker: `create()`, `lookup()`, and `delete()`.

The availability of callbacks poses the question of how to handle kernel resources acquired through them in the event that a plugin needs to be terminated. Because of the rich variety of kernel resources, we considered building a system that tracks all of them to be impractical. We believe that the runtime's owner service is able to handle the cleanup of the limited number of kernel resources it makes available to its plugins in a much more efficient way, if at all needed, e.g. through callback wrappers tracing resource usage, etc.

4.3 Memory Model

The memory model of the plugins' execution environment is influenced by the choice of hardware isolation mechanism. The scheme exploits features of the Intel x86 architecture's segmentation and protection hardware by placing all plugins into an unused privilege ring. While such hardware dependence may seem restrictive, the 'privilege rings' concept on which it relies is available on all modern CPU architectures. The most popular ones, Intel's IA32 and IA64, provide 4 privilege rings,

```
long create(runtime_t * rt, char * code, char * name);
long delete(runtime_t * rt, char * name);
long lookup(runtime_t * rt, char * name);
```

Figure 1: Built-in plugins' prototypes

```
long call_plugin(int id, runtime_t * rt, ...);
```

Figure 2: Gate function for invoking plugins

whereas others like the SPARC and the PowerPC provide only 2 privilege rings for supervisor and user mode, respectively. Kernel plugins can still be implemented on the latter in at least two different ways. One is to place plugins in pinned, unpaged memory in the user-level privilege ring. Isolation is enforced by the hardware and many overheads associated with using a process are avoided. Another option is to place plugins within the kernel privilege ring but to restrict them to dynamically generated code, thereby guaranteeing that they cannot interfere with paging and segmentation hardware. The former approach allows the use of arbitrary code in plugins at the expense of requiring somewhat complicated transfer of control between privilege rings. The latter approach invokes plugins just like ordinary kernel functions, but restricts them to dynamically generated code.

On x86 hardware, the OS kernel runs in ring-0 (highest-privilege). We allocate memory to hold all plugins' code, data, and stacks in ring-1, thereby guaranteeing the kernel memory's safety. In contrast, callback built-ins are invoked through a hardware trap, not unlike system calls, and run in ring-0, that is, they run in the OS kernel. Control and data flows between privilege rings are governed by the host kernel through hardware traps.

Plugins have full access to their parameters and local variables allocated on the plugin stack. They also have full access to a pool of ring-1 memory, effectively acting as a heap. The contents of the heap persist between plugin invocations, so it is also used for static variables. The heap is allocated on a per-runtime basis, which means that all plugins within a runtime share it and can use it for global variables, communication, and cooperation. Additionally, it is possible to provide select plugins with read-only access to parts of the kernel proper's memory. While such a feature could simplify the implementation of system monitoring plugins or the sharing of data between the kernel and plugins, it can also have security implications so it should be employed judiciously.

4.4 E-code Language Specification

In our design, plugins can be specified either as pre-compiled machine code, or in E-code – a language akin to 'C [19] and developed as part of the ECho high-performance event-delivery middleware [10]. E-code is a fairly complete subset of the C language that compiles to native machine code at runtime using a dynamic code generator that processes one function at a time. A more detailed list of E-code capabilities follows:

- *Datatypes:* E-code supports the following basic types: `char`, `int`, `float`, `double`, and `boolean`. It also supports structures, pointers (including pointers to structures), and pointer arithmetic.
- *Variables:* Global variables are allocated on the heap, which is a per-runtime pool of ring-1 memory persistent across plugin invocations. Local variables are allocated on the plugin stack.
- *Function calls:* Plugins are allowed to perform function calls only to other functions or callbacks registered within their runtime. Appropriate trap or trampoline code for the invocation is generated automatically and transparently.
- *Function prototypes:* Plugin functions must conform to a prototype convention – their first argument must be a '`runtime_t *`' to provide linkage back to their runtime. Furthermore, their result type is restricted to `long`, however, that is not a severe restriction since most basic datatypes are easily cast to a `long` value, with the notable exception of the class of floating point numbers, which must be passed back by reference.
- *Language:* E-code supports the C operators, `for` loops, `if`, and `return` statements.

Currently, E-code does *not* support `while` loops, `switch` statements, unions, and function pointers, though they do not pose conceptual difficulties and can be implemented if needed in the future.

4.5 Interface

The kernel/plugin interface consists of the runtime namespace manipulation routines, any additional kernel callbacks that an extensible subsystem instance exports to its plugins, the plugin invocation mechanism, and the pool of plugin static memory.

The runtime namespace manipulation routines displayed in Figure 1 are implemented as kernel proper functions. Thus, they are directly available to the kernel proper and are isolated from plugins, yet available to them in the form of 'built-in' plugins. Besides that mandatory minimum, each extensible service can augment the interface by exporting more kernel callbacks of its choosing, e.g. `sendmsg()` and `recvmsg()`, in the form of additional 'built-in' plugins. We continue with a more detailed description of the namespace manipulation interface.

Creation

Each plugin function is specified by a tuple that describes it completely. The tuple consists of the following elements:

- *Runtime pointer:* It refers to the runtime this function is to be created in. All functions' prototypes have a '`runtime_t *`' first argument serving as a link to their runtime environment and allowing them to interface with other functions. It provides closure (in the mathematical sense) of the namespace with respect to the operations of its functions.
- *Code:* This is either an ASCIIZ string specifying a single E-code function or a pre-compiled relocatable machine code dump. In the former case the runtime translates the E-code function into efficient, native machine code at creation time. The translation is a one-time cost and is amortized over all subsequent executions of that function. Translation costs are relatively small, thanks to the efficiency of E-code's dynamic code generator [10]. For example, the image grayscaling plugin used in our experimental evaluation consists of a 66-line E-code function which translates in only 4 mS, compared to the 700 mS it takes to spawn an external compiler (with compiler binary already present in the OS buffer cache).
- *Name:* A string constant providing the name this function is to assume in the runtime's symbol table. After its creation, a function can be looked up and called upon using that name.

Deletion

Deleting a function is a straightforward operation that deallocates the code and static data resources associated with it and then unlinks it from the symbol table. The `delete()` built-in plugin's prototype is self-explanatory and also appears in Figure 1.

Invocation

A function is available for execution immediately after its creation. The actual invocation, however, is not as trivial as a simple function call because of the privilege ring-based isolation scheme.

From Kernel Space: Normally, hardware does not allow higher-privileged code to call untrusted, lower-privileged code. To circumvent the problem our framework provides a 'gate' function `call_plugin()` that encapsulates the implementation complexity and hides hardware details. This makes invoking any plugin as simple as calling the gate function whose prototype is shown in Figure 2.

The gate function looks up the target plugin's entry in its runtime's symbol table and copies the declared number of parameters from the kernel's to the plugin's stack. It then invokes the plugin by branching to its address and sidestepping the hardware restriction. The mechanics of the latter are described in more detail in the implementation section.

From Plugin Space: To encourage function composition we provide a similar gate function in the isolated address space, permitting plugin functions to invoke each other. It is syntactically and semantically identical to its counterpart employed from the host kernel despite significant implementation differences.

Invoking a plugin function from another one has overhead akin to that of a simple function call. The reason for this being that control flows within the isolated address space and no protection boundary needs to be crossed. The benefit is that this enables plugin functions to cooperate easily and cheaply, thereby increasing the utility of the model for complex extensions.

Invoking a kernel callback (built-in plugin) from a dynamic, user-defined one, however, does require crossing the protection boundary from ring-1 back into ring-0. This is achieved by means of a hardware trap, the details of which are hidden in the gate function's implementation and explained further in the next section.

Finally, irrespective of whether the call originates in ring-0 or ring-1, invoking a plugin requires naming it unambiguously, i.e., by its name and runtime context. Unfortunately, matching name strings in the symbol table repeatedly is needlessly expensive. To avoid that overhead, we map the string name to an integer id unique within each runtime, thereby speeding-up lookup and simultaneously making id caching much easier. All built-ins are also assigned fixed well-known integer ids. The mapping between dynamic plugins' names and integer ids is performed by the `lookup()` plugin.

5 Implementation

A prototype of the kernel plugin facility has been implemented in recent stable-tree Linux kernels (versions 2.4.18 and 2.4.19). The prototype implements hardware isolation, dynamic code generation, and dynamic linking fully, though details of the design are still evolving. This section describes some relevant implementation details and their implications.

5.1 Hardware Fault Isolation

Our plugin isolation scheme is a clean re-implementation of a popular concept employed in a number of systems, both virtual machines and others, e.g. VMware [25], Palladium [8], etc. It exploits features of the segmentation and privilege checking hardware of the Intel x86 architecture to achieve

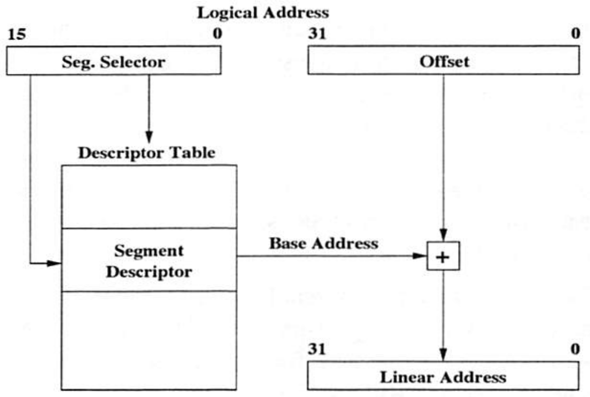


Figure 3: Intel x86: logical to linear address translation

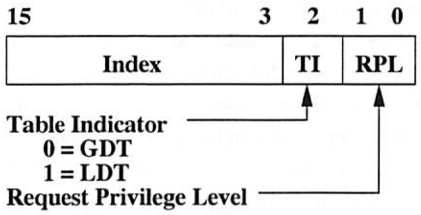


Figure 4: Intel x86: segment selector layout

address space isolation within the Linux kernel. We briefly describe the method next; for a full discussion, the reader is referred to the original papers and Intel documentation.

The fundamental idea is to allow application-specific code to run in the core kernel by placing it in a separate protection domain and relying on hardware to enforce it. The new domain is an address space – a proper subset of the Linux kernel's virtual address space. While the kernel itself can access the plugins' address space freely, plugins cannot, in general, access the larger kernel memory.

Segmentation

In practice, domains are implemented as protected memory segments directly supported by the hardware MMU. Segments are ranges of consecutive addresses described by base address and length. The operating system maintains linear tables of 'segment descriptors' for all segments. In addition to base and length, each segment descriptor stores privilege, access, and type information for its segment. Two tables of descriptors are active at any given point in time: the Global Descriptor Table (GDT) and a Local Descriptor Table (LDT). While the former is static and immutable, the latter is a per-process structure.

Figure 3 depicts how segmentation addressing works on the x86. Logical addresses are composed of a 16-bit segment selector and a 32-bit offset. Segment selector values are used to index into segment descriptor tables. The layout of a segment selector is shown in Figure 4.

It contains a linear table index, a one-bit Table Indicator (TI), and a two-bit Requested Privilege Level (RPL). The TI determines which of the two currently active tables the selector is referring to, whereas the RPL is used in privilege checks. The chosen descriptor entry provides a base address, a limit to check the offset against, read/write/execute permissions, and a descriptor privilege level (DPL).

The CPU maintains a Current Privilege Level (CPL) for the currently executing instruction. The DPL is compared to the CPL and RPL for each memory access. In general, CPL and RPL need to be numerically less than or equal to the target segment's DPL (i.e. at the same or higher privilege) in order for access to be granted. For a complete explanation of access permission checks and motivation for the existence of the RPL the reader is referred to Intel documentation [1, pp. 105–140]. All access violations, such as pointer dereferences to memory outside of the ring-1 segment (including NULL pointer dereference), attempts to execute an illegal or protected instruction, etc., result in an exception being thrown and a trap to the kernel proper to handle it. Additionally, the owner service allows each plugin a quantum of time in which to complete. Overrunning that quantum is detected by a periodic timer interrupt and also results in forceful preemption and a trap to the kernel.

Our current method for dealing with offending plugins is immediate termination. An interesting future direction we are considering is to implement a recovery mechanism, allowing extensible services the choice to terminate or to continue a plugin based on custom per-service policies and the type of the failure or misbehavior.

Initialization

At boot time, the plugin facility allocates a region of memory to be used exclusively for plugin code and data. A simple first-fit private memory allocator is initialized with the parameters of the pool and is used to allocate memory for structures to be placed in the isolated area.

Two new segment descriptors are computed and installed in the GDT, each covering the whole isolated memory pool. Both descriptors are assigned the same ring-1 privileges but different types. The type of the first one is set to 'code' and it is used to address executable plugin code. The type of the second one is set to 'data', and it is used for data manipulations involving plugin stacks and heaps. This simple overlay scheme was chosen to ease the initial implementation effort by avoiding the need to parameterize the memory allocator for disjoint code and data memory pools. It could be replaced with a split code and data design in the future, to prevent the possibility for self-modifying plugin code and to limit the amount of damage a misbehaving plugin can wreak upon other plugins.

A third segment descriptor can be defined overlaying part or all of the kernel's ring-0 memory but accessible in read-only mode from ring-1. Although such a segment has the potential of simplifying or optimizing kernel/plugin data interactions it has to be used with great care because of possible security implications. It can be thought of as an optional feature for cases when performance benefits outweigh potential security concerns.

Runtimes are created at the request of kernel services. Each runtime's control structure contains pointers to its stack and heap as well as a symbol table of registered plugins. The control structure is allocated in ring-0 kernel memory to protect it from being tampered with by plugins. The built-in plugins are implemented as trusted kernel subroutines enabling them to modify the control structures during operations like creation or deletion of dynamic plugins. In contrast, the heap, stack, and code of plugins are all allocated within ring-1 memory.

Control Transfers

Passing control from a plugin in ring-1 to the kernel in ring-0 is straightforward, by use of a trap gate similar to the one implementing system calls from user-space (ring-3). The hardware handles the trap and passes control to the kernel in a protected fashion. Any state needed for returning to ring-1 is saved on the kernel stack. If the control transfer is to be one way (e.g. return from a plugin) rather than two way (e.g. kernel callback), then that state is simply cleaned up by the kernel trap handler.

Passing control from the kernel to a plugin, unfortunately, is more difficult. There is an inherent asymmetry in control transfers between privilege levels because the hardware is designed to prohibit high-privileged code from invoking lower-privileged code. To sidestep the problem, a stack frame is carefully forged (on the ring-0 stack) that emulates the state the stack would have had if it had been called from ring-1 and then executes a `ret` instruction to the forged return address. This causes the CPU to switch into ring-1 and start execution of the targeted plugin function.

Interrupts

x86 interrupt handlers run in ring-0. In case an interrupt occurs while the CPU is already executing in kernel mode, the interrupt simply grows the current stack. If, however, the CPU is executing in a privilege level different from ring-0, then it switches to ring-0 immediately, performing the necessary stack swap to the *bottom* of the kernel stack. This behavior is predicated on the premise that an interrupt occurring while in user-space has no kernel state to preserve, so the new frame can start from the base of the kernel stack.

It is not hard to see how this otherwise normal behavior can cause trouble when interacting with ring-1 plugins, however. The aforementioned premise is negated because plugins *are*, in effect, a part of the kernel, yet they execute outside of ring-0. If an interrupt fires while a plugin is running, the CPU switches immediately to a ring-0 handler and uses the kernel stack. Unfortunately, starting from the *base* of the stack it overwrites the state already there, accumulated prior to invoking the plugin. This effect is due to the unconventional use of privilege rings to implement what amounts to protected upcalls of which the hardware is unaware.

There are a few possible solutions to this problem: (1) disable interrupts while plugins are running, (2) save and restore the kernel stack before and after plugin invocations, and (3) trick the hardware to *grow* the stack upon an interrupt in ring-1.

While the first solution does not add any overhead to plugin execution, it has the undesirable effect of blocking interrupts for potentially non-trivial lengths of time. In the kernel, blocking of *all* interrupts is allowed only for the shortest times, since it could lead to loss of important device interrupts and disrupt the operation of peripherals. Moreover, such a solution would also prevent the implementation of plugin preemption, which relies on a periodic hardware timer interrupt. Clearly, this approach is unsuitable.

The second solution, saving the kernel stack's state before plugin invocation and restoring it immediately after that, is workable. It was our first implementation, but it increased plugin invocation overheads and introduced significant irregularities in their cost, due to the unpredictable amount of state (up to a page frame in the worst case) that needs to be saved and restored each time.

These disadvantages led us to come up with our final solution. It is based on an architectural programming trick that fools the interrupt handling hardware into *growing* the kernel stack rather than overwriting its bottom, despite the fact that the interrupt occurs outside of ring-0. The trick involves careful manipulation of the stack base pointer in the task state segment structure (TSS) of the CPU [1]. This allows us to continue servicing interrupts while plugins are running, yet, at the same time avoid the unpredictability of kernel stack saving and restoring. As an added bonus, the overhead of this method is extremely small, its implementation consisting of only a few assembly instructions.

A Remaining Issue

A discussion of kernel plugins would not be complete without mention of any remaining issues with their current implementation. One such issue is the lack of protection across multiple plugins. Thanks to the compartmentalization of each client's plugins into a separate runtime, plugins have no means of naming symbols in other runtimes. This, however, does not provide firm isolation guarantees, even though it raises the bar for how difficult it would be for a plugin to interfere with plugins in other runtimes.

To address this issue, we are considering developing our scheme further to include two GDT descriptors *per runtime*, to describe each runtime's code and data separately from other runtimes. In this way, we can exploit the segmentation hardware further and achieve isolation not only between the kernel proper and plugins but also among runtimes. Such an enhancement would not add any runtime overhead and is under active development.

5.2 Dynamic Linking

As part of the provided trusted runtime environment, a dynamic linker operates on a runtime's symbol table and implements symbol creation, lookup, execution, and deletion. The symbol table is an array of symbol structures, and looking a symbol up in it has linear complexity. This choice was made to simplify the initial implementation and will not result in problems unless a very large number of plugins are registered within a single runtime. Re-coding the symbol table as a hash table may be used to address this issue should it become necessary.

5.3 Machine Code Generation

The E-code dynamic code generator operates by parsing the source language and emitting the appropriate instructions into a memory buffer from which they can be executed directly. Currently, E-code supports dynamic code generation for Intel x86, MIPS, StrongARM, and Sun SPARC (32- and 64-bit) processors. Support for Intel's 64-bit EPIC architecture is under development.

The code generator is subroutine-based and does not require invocation of external binaries. Code is emitted during parsing in the form of virtual instructions for an idealized RISC architecture. Simple, low-hanging fruit optimizations are applied (constant propagation, register renaming, and limited common subexpression elimination), and then the virtual instructions are mapped to their physical counterparts for the target architecture.

E-code's early versions were based upon Icode, an internal interface developed at MIT as part of the 'C project [19]. Icode is itself based on Vcode [11], also developed at MIT by Dawson Engler. E-code's recent versions, however, are based on DRISC, a low-level DCG package developed at Georgia Tech. The performance of the two versions are similar. More information about characteristics of the E-code language, such as examples of the generated code, further details on its performance, and generation times can be found elsewhere [10].

6 Experimental Evaluation

This section demonstrates the base performance of plugins using two micro-benchmarks, two macro-

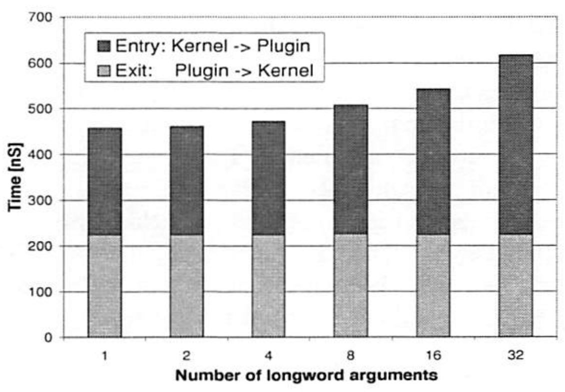


Figure 5: Control transfer cost vs. number of arguments

benchmarks, and finally, an evaluation of the utility and practicality of plugins with a realistic image-transcoding plugin. The latter color-downsamples images using an application-specific integer-only method as per some client's needs.

Experimental results reported in this section are obtained on an 866 MHz Pentium III processor, with 16 KB L1 I&D caches, 256 KB unified L2 cache, 512 MB of PC133 RAM, and a 20 GB Western Digital WD205AA hard drive used in UDMA66 mode. The operating system is Fedora Core 1 running Linux kernel version 2.4.19 augmented by our kernel plugin facility.

Timing is performed using the Pentium processor's internal time-stamp counter (TSC), except for the *httperf* [17] macro-benchmark which uses much coarser granularity system timers. The TSC is a 64-bit register zeroed at power-up and incremented by exactly 1 with each clock tick of the CPU core. Its 2 nS error is insignificant in comparison to statistical variations in experimental data.

6.1 Micro-benchmarks

Plugin Execution

Two metrics important to any server application are latency and throughput. We measure the impact that placing code in a kernel plugin has on these metrics. We define *latency overhead* as the amount of time passing between the first instruction of a kernel plugin invocation and the execution of the plugin's first instruction. Latency overhead thus defines the latency cost of utilizing the kernel plugin facility. Similarly, we define *throughput overhead* to be the execution time of a null plugin. This represents the pure cost of the plugin abstraction. Since plugins are executed directly on the underlying hardware, these metrics are the only runtime costs incurred by kernel plugins.

Micro-benchmark data displayed in Figure 5 depicts the execution time of a null kernel plugin (in nanoseconds), versus the number of long word arguments passed to it. Execution time is comprised of two parts: entry into the plugin and exit from the plugin. The first part characterizes the latency overhead experienced due to plugin use, whereas the second part represents the remaining cleanup overhead at exit time. It is easily observed from the graph that the entry latency is weakly linearly dependent on the number of plugin parameters, whereas the exit overhead is constant. It is important to realize that the measurements in Figure 5 are for plugin invocation from the kernel. Function invocation from within ring-1 is almost identical to a user-space function call, meaning that it is essentially 'free'.

Results are obtained by timing 2001 runs, dropping the first one to avoid cold CPU cache effects and averaging the rest. Furthermore, interrupts are disabled during each individual benchmark run to shield measurements from the high timing variability that interrupts induce under Linux v2.4 [14]. The observed standard deviation for this plot is less than 1% from the mean, implying very high confidence in the data and predictability of the mechanism's performance.

The main result is that for a reasonable number of parameters, the baseline cost of kernel plugins is between 0.45 μS and 0.62 μS for this hardware. Thus, our implementation's performance is on par or better than similar schemes [8] (after adjusting for our faster hardware) despite major differences in kernel architecture (2.0.34 vs. 2.4.19) and implementation methodology.

Plugin Creation/Deletion

Since kernel plugins are intended to be easily and frequently updated, it is important to characterize their creation and deletion costs.

Our dynamic code generator uses subroutine-based techniques that do not require invocation of an external compiler. It is fast and of time complexity roughly linearly proportional to the source code size. For the sample image-transcoding plugin used in our experiments the costs for code generation, linking, and unlinking of the plugin are 4 mS, 3.1 μS, and 1.6 μS respectively.

6.2 Macro-benchmarks

To better understand application-level effects of the baseline costs associated with different isolation techniques and with kernel plugins in particular, we next turn our attention to macro-benchmarks. We compare and contrast the overheads and service effects of two proposed isolation techniques for user-specific kernel extensions: placing extensible services in virtual machines, and implementing extensions as kernel plugins.

Modern services need to provide customizability while maintaining high levels of performance. Generally these two imperatives are in conflict. For our particular exper-

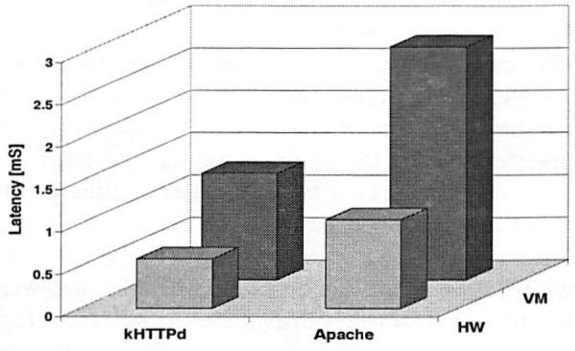


Figure 6: Latency effects of commercial VM technology (VMware 3.2.0)

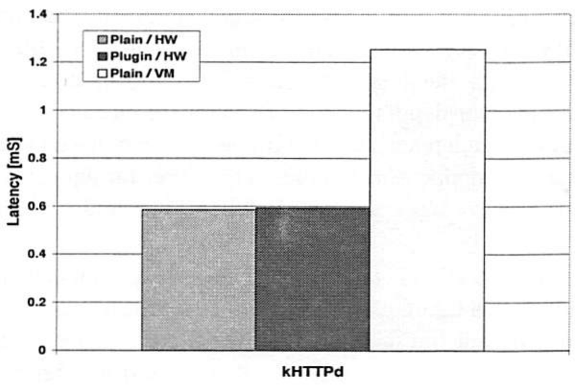


Figure 7: Comparative latency effects of kernel plugins vs. commercial VM technology

iments, we chose to look at a web server, as it is a relatively simple, typical, and popular service with many available implementations. A well-known approach to building high-performance web servers is to run the service daemon within the OS kernel. While this eliminates many inefficiencies inherent to user-space and increases the performance of the server substantially, it has the unfortunate effect of discouraging extensibility due to safety concerns when running within the kernel. We propose to use kernel plugins to rectify this problem.

To quantify our assertions and to provide a better gauge for the expected performance of typical kernel- vs. user-space web servers, as well as different isolation techniques, we measured the server reply latency and reply throughput of popular web server implementations and report our findings in figures 6, 7, and 8.

The web servers ran on the machine described previously, while the test load was provided by a more powerful Dell Workstation 340 (2.2 GHz Pentium 4, 512 KB L2 cache, 512 MB of RDRAM-400) over an otherwise quiescent 100 Mbps Ethernet network. Results for these three figures were measured in user-space at the client

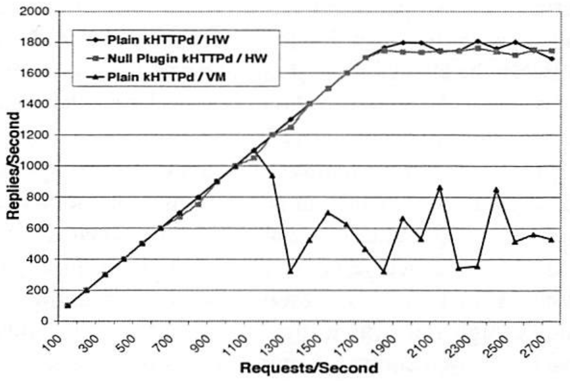


Figure 8: Isolation technology impact on throughput

workstation and include jitter induced by interrupt processing on both ends. Therefore, these are a good indicator of subjective performance for a client application in this environment.

Figure 6 provides a server reply latency comparison between kernel- vs. user-space based servers, as well as a measure of the cost of full virtualization (as measured on an industry standard product VMware 3.2.0). The figure shows that a typical kernel-space web server's latency (kHTTPd) is roughly half of a user-space server's latency (Apache). It also shows that full virtualization increases service latency 2-2.5 times. Thus, the addition of safe extensibility through virtual machine techniques likely cancels the performance benefits of employing kernel-based web servers.

Our proposed alternative, kernel plugins, compare favorably latency-wise to the baseline case and the VM solution, as shown in Figure 7. We measure an unmodified kernel web server (baseline), the same server extended with a null kernel plugin (invoked once per request from within kHTTPd), and another copy of the server isolated in a VMware virtual machine. Kernel plugins add minute latency overhead to the service being extended. Figure 7 shows an average plugin overhead of 8 μS, though that value is inflated due to the well-known significant variability that interrupt processing induces in Linux [14]. A comparison between the averages of the top 10% timing samples provides a less variable and more accurate estimate of the real plugin overhead (less than 1 μS), consistent with the jitter-free results of Figure 5. The small cost is striking compared to the larger latency overhead imposed by the VM approach. The benefit obtained by using plugins over virtual machines comes from achieving extensibility without complete virtualization. Instead, only the isolation properties of virtualization are really needed. Since kernel plugins are designed to provide exactly that, they are able to avoid unnecessary overhead.

The last macro-benchmark we consider explores the server's throughput degradation as a function of the isolation technique. We define throughput as the relation between the clients' request rate and a server's sustained reply rate. We measure throughput utilizing the well-known *httperf* benchmark [17].

Figure 8 shows a family of graphs describing the throughput performance of an unmodified kernel-based web server (baseline), a null-plugin modified server, and an unmodified server running in a virtual machine. While kernel plugins preserve the server's ability to handle high throughput almost untouched, the virtual machine-based server is saturated at a little more than half the throughput. Similarly, consistency and predictability inside a virtual machine are severely degraded. In contrast, the kernel plugin approach is remarkably stable and consistent. Again, the disparity is attributed to the many unnecessary non-isolation-related aspects of full virtualization.

We note that the inefficiencies inherent in virtualization schemes like VMware's are known and that there are promising alternatives like Xen [3]. However, Xen requires a host OS kernel to be ported to its specially defined abstract VM model, whereas plugins can be used with existing operating system kernels. The Xen approach, therefore, is complementary to our research.

6.3 A Practical Example

To demonstrate the utility and actual real-life performance of plugins we provide a practical example of their use. Specifically, we compare and contrast the performance of a user-space and a kernel-space web server, both with and without extensions. The user-space web server is the popular Apache (version 2.0.48). We extend Apache with an image-transcoding function that reduces image color-depth from 24-bit true-color to 8-bit monochrome. It is an example of a useful extension that a PDA with modest display, CPU, and power resources, could use to adapt images to its capabilities and/or to shift workload to the server. For a kernel-space web server, we use kHTTPd which comes standard with stock Linux v2.4 kernels. We extend kHTTPd with the same color-depth reduction code, placing it into the kernel both as an unprotected kernel function and as a dynamically deployable, isolated kernel plugin. The transcoding function consists of 66 lines of E-code including whitespace and comments and compiles to 371 instructions totaling 1078 bytes of machine-code. For comparison, gcc 3.2.2 without optimizations compiles the same code into 245 instructions totaling 623 bytes, and -O2 optimizations shrink that further to 151 instructions and 338 bytes. Despite its larger size E-code machine-code has roughly similar code path length as unoptimized gcc code, determined by hand comparison of the resulting machine-code. The absolute size difference is a consequence of E-code's simpler but faster code generation strategy.

Experiments consist of repeatedly requesting images from the web servers and recording request service times (measured at the server side). Timing instrumentation is implemented using the Pentium time-stamp counter, and again 2001 samples are taken, disregarding the first to control against cold OS buffer-cache effects. The images we used were in the Portable Pixmap (PPM) format with sizes 9 KB, 99 KB, 270 KB, and 3.3 MB. The sizes were chosen to approximate both extremes, as well as average typical online image sizes. Most image data on the Internet today is encoded in the JPEG format, which is highly compressed and harder to transcode than the relatively simpler PPM format. To avoid the graphics complexity, yet account for the format differences, we emulate typical JPEG file sizes with the thumb, small, and medium PPM data sets. To emulate the pixel dimension of JPEG files we use the large PPM data [7]. Moreover, note that during color-depth reduction PPMs are reduced by 66%, because each pixel's RGB components are replaced with a single monochrome value. Therefore, the processed images have sizes of 3 KB, 33 KB, 90 KB, and 1.1 MB, respectively.

Figures 9, 10, 11, and 12 present our experimental results. Each figure plots service times for the servers with and without transcoding. The first item to note is the oscillation in the performance of Apache from Figure 9 to Figure 10, as opposed to the performance of kHTTPd. The reason for this oscillation is that the transcoding plugin touches the contents of the entire file during conversion and the time spent transcoding exceeds the time saved by bandwidth reduction for small files. In contrast, kHTTPd does not exhibit such oscillation, despite identical transcoding size reductions. We believe that this is due to a combination of factors related to efficiency gained from co-location in the kernel: (1) avoiding multiple user/kernel protection boundary crossings, (2) related reduction in data copying, (3) benefits from kernel code non-preemptability in Linux, (4) related improvement in CPU cache and TLB performance. In essence, the overhead of reading data from disk should dominate this benchmark, but once the data is in memory (after the OS buffer cache has warmed up), the co-located in-kernel transcoding and the asynchronous network send cost relatively little when compared to their counterparts in user-space, which are further subject to scheduling.

The minimal difference between transcoding costs incurred by the unprotected kernel function and the dynamically deployed kernel plugin suggest that plugins are on the same order of latency. In practice today, function invocations are considered to be essentially 'cost-free'. We view the fact that kernel plugins' costs are comparable as a validation for our design's achievement

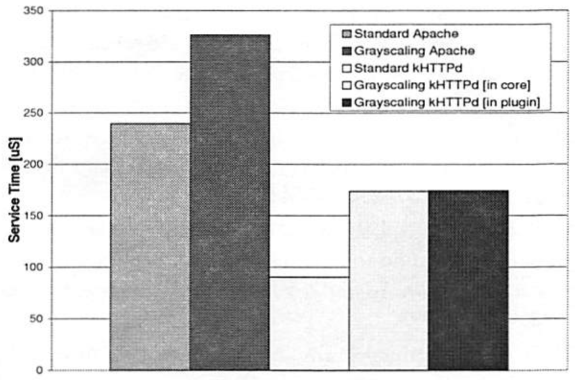


Figure 9: Service times for a thumb image (9KB)

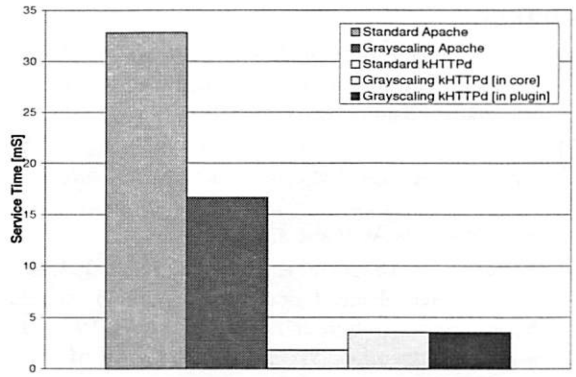


Figure 11: Service times for a medium image (270KB)

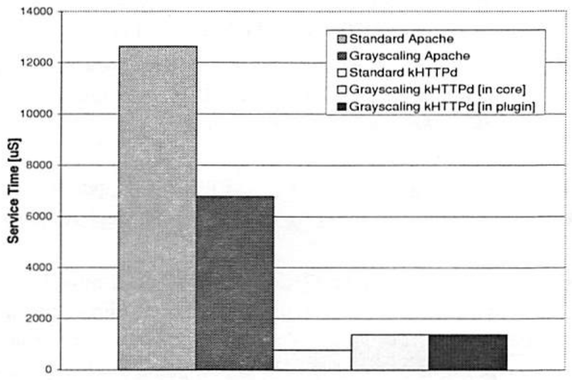


Figure 10: Service times for a small image (99KB)

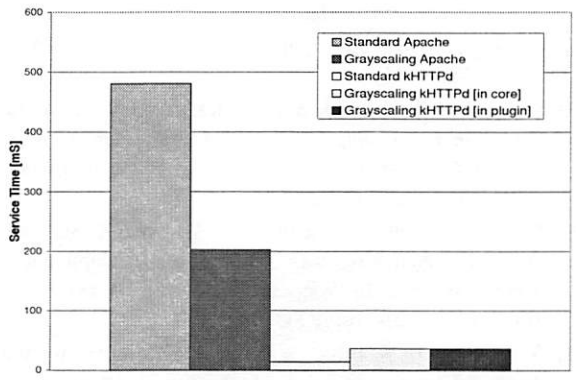


Figure 12: Service times for a large image (3.3MB)

of its efficiency and performance goals.

To summarize, experimental evaluation shows that kernel plugins enable applications to adapt kernel services and extract significant flexibility advantages, while being sufficiently lightweight to not compromise the gains from co-location in the kernel.

7 Conclusions and Future Work

We have presented the design, implementation, and evaluation of a novel framework for safe deployment of application-specific code into an OS kernel. The mechanism is based on three key technologies: hardware fault isolation, dynamic code generation, and dynamic linking. HFI relies on commonly available hardware features, and offers low-overhead isolation. Our dynamic code generation is based on E-code, a DCG package developed at Georgia Tech. Using DCG, plugins may be comprised of user-defined code, thereby enabling arbitrary application-specific specializations of the kernel services with which they are associated. Dynamic linking enforces a narrow kernel/plugin interface, provides logical isolation between extensible system-level entities, and eliminates kernel namespace pollution.

Micro-benchmarks evaluating kernel plugins show the base cost of plugin invocation to be between 0.45 μS and 0.62 μS. Plugin code generation, linking, and unlinking costs are 4 mS, 3.1 μS and 1.6 μS, respectively, for the sample image-transcoding plugin used in this paper. In general, code generation cost depends on code size, and both linking and unlinking costs can be improved further by optimization of the symbol tables currently used in the plugin facility. More importantly, macro-benchmarks and experimental results from a realistic sample application showcase performance advantages offered to end-user applications using kernel plugins in lieu of specializations implemented at user level.

In its current state, the plugin facility fully implements hardware fault isolation, dynamic code generation, dynamic linking, and plugin preemption based on hardware system timers. Planned future work and improvements include tighter integration of code generation and isolation, further performance characterization, exploration of inter-plugin memory protection, implementation of a fault recovery and continuation mechanism, porting the system to Intel's 64-bit Itanium 2 architecture, and optimization of the implementation bottlenecks.

References

[1] *Intel Pentium Processor Family Developer's Manual.* Volume 3: Architecture and Programming Manual. Intel Corporation, Santa Clara, CA, 1995.

[2] A. Banerji and D. L. Cohn. An infrastructure for application-specific customization. In *Proceedings of the 6th Workshop on ACM SIGOPS European workshop*, pages 154–159. ACM Press, 1994.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the* 19^{th} *Symposium on Operating Systems Principles*. ACM Press, October 2003.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pages 267–283. ACM Press, 1995.

[5] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of the* 5^{th} *International Conference on Open Architectures and Network Programming*, pages 141–152. IEEE, 2002.

[6] F. E. Bustamante, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active streams: An approach to adaptive distributed systems. In *Proceedings of the* 8^{th} *Workshop on Hot Topics in Operating Systems*, 2001.

[7] S. Chandra, C. S. Ellis, , and A. Vahdat. Differentiated multimedia web services using quality aware transcoding. In *INFOCOM 2000 - Nineteenth Annual Joint Conference of the IEEE Computer And Communications Societies*, March 2000.

[8] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the* 17^{th} *ACM Symposium on Operating Systems Principles*, pages 140–153. ACM Press, 1999.

[9] Connectix, Corp. The technology of Virtual PC, 2000.

[10] G. Eisenhauer, F. E. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. *ACM SIGOPS Operating Systems Review*, 35(2):7–20, July 2001.

[11] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170. ACM Press, 1996.

[12] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pages 251–266. ACM Press, 1995.

[13] A. Gavrilovska, K. Mackenzie, K. Schwan, and A. McDonald. Stream handlers: Application-specific message services on attached network processors. In *Proceedings of the* 10^{th} *Symposium on High Performance Interconnects*, August 2002.

[14] A. C. Heursch, A. Horstkotte, and H. Rzehak. Preemption concepts, Rhealstone benchmark and scheduler analysis of linux 2.4. In *Proceedings of the Real-Time & Embedded Computing Conference*, November 2001.

[15] J. Liedtke. On μ-kernel construction. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.

[16] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, 1993.

[17] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, June 1998.

[18] G. C. Necula. Proof-carrying code. In *Proceedings of the* 24^{th} *Annual Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A template-based compiler for 'C. In *Proceedings of the First Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[20] M. Satyanarayanan and C. S. Ellis. Adaptation: the key to mobile I/O. *ACM Computing Surveys (CSUR)*, 28(4es):211, 1996.

[21] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the* 2^{nd} *USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227. ACM Press, 1996.

[22] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[23] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the* 3^{rd} *Symposium on Operating Systems Design and Implementation*, pages 117–130. USENIX Association, 1999.

[24] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[25] VMware, Inc. VMware virtual platform, technical white paper, 1999.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the* 14^{th} *ACM Symposium on Operating Systems Principles*, pages 203–216. ACM Press, 1993.

[27] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: application-specific handlers for high-performance messaging. *IEEE/ACM Transactions on Networking (TON)*, 5(4):460–474, 1997.

[28] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation

Ian Piumarta

Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 4, place Jussieu, 75252 Paris Cedex 05, France

ian.piumarta@inria.fr

Abstract

Tools supporting dynamic code generation tend too be low-level (leaving much work to the client application) or too intimately related with the language/system in which they are used (making them unsuitable for casual reuse). Applications or virtual machines wanting to benefit from runtime code generation are therefore forced to implement much of the compilation chain for themselves even when they make use of the available tools. The VPU is an fast, high-level code generation utility that performs most of the complex tasks related to code generation, including register allocation, and which produces good-quality C ABI-compliant native code. In the simplest cases, adding VPU-based runtime code generation to an application requires just a few lines of additional code—and for a typical virtual machine, VPU-based just-in-time compilation requires only a few lines of code per virtual instruction.

1 Introduction

Dynamic compilation is the transformation of some abstract representation of computation, determined or discovered *during* application execution, into native code implementing that computation. It is an attractive solution for many problem domains in which high-level scripts or highly-compact representations of algorithms must be used. Examples include: protocol instantiation in active networks, packet filtering in firewalls, scriptable adaptors/interfaces in software object busses, policy functions in flexible caches and live evolution of high-availability software systems. In other domains it is an essential technique. In particular, virtual machines that interpret bytecoded languages can achieve good performance by translating bytecodes into native code "on demand" at runtime.

A dynamic code generator is the part of a dynamic compiler that converts the abstract operations of the source representation into executable native code. It represents a large part (if not the bulk) of any dynamic compiler—and certainly the majority of its complexity. However, very few utilities exist to ease the task of creating a dynamic code generator and those that are available are ill-suited to a simple, "plug-and-play" style of use.

1.1 Related work

Several utilities have been developed to help with the implementation of static code generators. Examples include C-- [1], MLRISC [3] and VPO [2] (part of the Zephyr compiler infrastructure). However, very few tools have been developed to help with the implementation of dynamic code generators. Those that do exist are concerned with the lowest (instruction) level of code generation.

`ccg` [9, 12] is a collection of runtime assemblers implemented entirely in C macros and a preprocessor that converts symbolic assembly language (for a particular CPU) embedded in C or C++ programs into macros calls that generate the corresponding binary instructions in memory. It is useful tool for building the final stage of a dynamic code generator, but constitutes only a small part of a dynamic compiler, and deals exclusively in the concrete instructions of a particular architecture.

`vcode` [6] and GNU Lightning [13] are attempts to create virtual assembly languages in which a set of "typical" (but fictitious) instructions are converted into native code for the local target CPU.[1] Both of these systems present clients with a register-based abstract model. Register allocation (one of the most

[1] GNU Lightning is based on `ccg` and is little more than a collection of wrappers around it.

difficult code generation problems to solve) is left entirely to the client, and both suffer from problems when faced with register-starved, stack-based architectures such as Intel.[2]

Beyond these systems code generators rapidly become intimately tied to the source language or system with which they are designed to operate. Commercial Smalltalk and Java compilers, for example, use sets of "template" macros (or some functionally equivalent mechanism) to generate native code, where the macros correspond closely to the semantics of the bytecode set being compiled. Adapting them for use in a different application (or virtual machine) requires significant work. Tasks such as register allocation also tend to be adapted for the specifics of the source language. (One notable exception is the Self compiler [4], for which a fairly language-neutral model of stack frames and registers was developed. Nevertheless, the rest of the Self compiler is so complex that nobody has managed to extract and reuse it in a completely different context.)

1.2 The VPU

The VPU fills the gap between trivial "virtual assembly languages" and full-blown dynamic compilers intimately tied to their source language and its semantics. It is a complete "plug-and-play" dynamic code generator that can be integrated into any application in a matter of minutes, or used as the backend for a dynamically-compiled language or "just-in-time" compiler. It presents the client with a simple, architecture-neutral model of computation and generates high-quality, C-compatible native code with a minimum of time and space overheads. It assumes full responsibility for many of the difficult tasks involved in dynamic code generation, including register allocation and the details of local calling conventions. Applications and language implementations using the VPU are portable (with no source code changes) to all the platforms supported by the VPU; currently PowerPC, Sparc and Pentium.

A useful analogy might be to consider languages that are "compiled" into C source code which is then passed to an existing C compiler for conversion into the final executable. The VPU could be used in a similar fashion: its input "language" is semantically equivalent to C and its output is C-compatible native code. The difference is that the VPU is integrated into the application and performs its compilation at runtime, sufficiently fast that the application should never notice pauses due to dynamic code generation.

[2]GNU Lightning solves the "Pentium problem" by supporting just 6 registers. vcode solves the problem by mapping "excess" registers within its model onto memory locations, with all the performance penalties that this implies.

```
#include <stdio.h>
#include <stdlib.h>

typedef int (*pifi)(int);

pifi rpnCompile(char *expr);

int main()
{
  int i;
  pifi c2f= rpnCompile("9*5/32+");
  pifi f2c= rpnCompile("32-5*9/");
  printf("\nC:");
  for (i = 0;  i <= 100;  i += 10)
    printf("%3d ", i);
  printf("\nF:");
  for (i = 0;  i <= 100;  i += 10)
    printf("%3d ", c2f(i));
  printf("\n\nF:");
  for (i = 32;  i <= 212;  i += 10)
    printf("%3d ", i);
  printf("\nC:");
  for (i = 32;  i <= 212;  i += 10)
    printf("%3d ", f2c(i));
  printf("\n");
  return 0;
}
```

Figure 1: Temperature conversion table generator. This program relies on a procedure `rpnCompile()` to create a native code functions converting degrees Farenheit to Celsius and vice-versa.

The rest of this paper is organised as follows: Section 2 describes the feature set and execution model of the VPU from the client's point of view. Section 3 then describes in some detail the implementation of the VPU, from the API through to the generation of native code (and most of the important algorithms in between). Section 4 presents a few performance measurements, and finally Section 5 offers conclusions and perspectives.

2 Plug-and-Play code generation

A simple (but complete) example illustrates the use of the VPU. Figure 1 shows a program that prints temperature conversion tables. It relies on a small runtime compiler, `rpnCompile()`, that converts an input expression (a string containing an integer function in reverse-polish notation) into executable native code. The program first compiles

```
#cpu pentium

pifi rpnCompile(char *expr)
{
  insn *codePtr= (insn *)malloc(1024);
  #[  .org    codePtr
      pushl   %ebp
      movl    %esp, %ebp
      movl    8(%ebp), %eax
  ]#
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) #[
!       expr += n - 1;
        pushl   %eax
        movl    $(atoi(buf)), %eax
    ]#
    else if (*expr == '+') #[
        popl    %ecx
        addl    %ecx, %eax
    ]#
    else if (*expr == '-') #[
        movl    %eax, %ecx
        popl    %eax
        subl    %ecx, %eax
    ]#
    else if (*expr == '*') #[
        popl    %ecx
        imull   %ecx, %eax
    ]#
    else if (*expr == '/') #[
        movl    %eax, %ecx
        popl    %eax
        cltd
        idivl   %ecx, %eax
    ]#
    else
      abort();
    ++expr;
  }
  #[    leave
        ret     ]#;
  return (pifi)codePtr;
}
```

Figure 2: Low-level code generation based on macro expansion. The input string is scanned for literals and arithmetic operators, and Intel native code generated accordingly. The sections delimited by '#[...]#' describe the code to be generated (and also "group" their contents like C curly braces). A preprocessor converts these sections into macro calls that produce the binary instructions in memory.

```
#include "VPU.h"

pifi rpnCompile(char *expr)
{
  VPU *vpu= new VPU;
  vpu                          ->Ienter()->Iarg()
                               ->LdArg(0);
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) {
      expr += n - 1;
      vpu                      ->Ld(atoi(buf));
    }
    else if (*expr == '+')  vpu->Add();
    else if (*expr == '-')  vpu->Sub();
    else if (*expr == '*')  vpu->Mul();
    else if (*expr == '/')  vpu->Div();
    else
      abort();
    ++expr;
  }
  vpu                          ->Ret();
  void *entry= vpu             ->compile();
  delete vpu;
  return (pifi)entry;
}
```

Figure 4: VPU-based code generation.

two temperature conversion functions `c2f` and `f2c`, and then uses them to produce a conversion table. The implementation of the "dynamic compiler" is encapsulated entirely within the `rpnCompile()` procedure.

Figure 2 shows one possible implementation of the `rpnCompile()` procedure. It uses `ccg` to create Intel native code for a function implementing the input expression. (The program fragments shown in Figures 1 and 2 combine to form a complete program, whose output is shown in Figure 3.)

Figure 4 shows an alternative implementation of `rpnCompile()` that uses the VPU to produce the same native code in a platform-independent manner. The VPU appears to clients as a single C++ class. A client constructs an instance of this class on which it invokes methods corresponding to the operations of an abstract stack machine. Once a complete function has been described the client asks the object to compile it. The result is the address of the native code implementing the function which can be subsequently be called from both statically- and dynamically-compiled code, just like a "normal" 'pointer to function'. When the function is no longer needed the client can return the memory to the heap using the standard C library function `free()`. Figure 5 shows the code generated by the VPU-based `rpnCompile()` procedure when run on the PowerPC.

Note that the only client-side state is the `vpu` instance itself; this would be the case no matter how complex the function being compiled. All instruction arguments (such as the constant argument for the `Ld` instruction) can be computed at runtime.

```
C:  0  10  20  30  40  50  60  70  80  90 100
F: 32  50  68  86 104 122 140 158 176 194 212

F: 32  42  52  62  72  82  92 102 112 122 132 142 152 162 172 182 192 202 212
C:  0   5  11  16  22  27  33  38  44  50  55  61  66  72  77  83  88  94 100
```

Figure 3: Output generated by the temperature conversion program.

```
-------------------------- code gen
003b14f0  mr      r4,r3
003b14f4  li      r5,9
003b14f8  mullw   r4,r4,r5
003b14fc  li      r5,5
003b1500  divw    r4,r4,r5
003b1504  addi    r3,r4,32
003b1508  blr
-------------------------- 28 bytes
-------------------------- code gen
003b1560  mr      r4,r3
003b1564  addi    r4,r4,-32
003b1568  li      r5,5
003b156c  mullw   r4,r4,r5
003b1570  li      r5,9
003b1574  divw    r3,r4,r5
003b1578  blr
-------------------------- 28 bytes
```

Figure 5: Generated code for PowerPC.

2.1 The VPU's model of computation

Clients are presented with a stack-based module of computation that includes a complete set of C operations: arithmetic and shift, type coercion, bitwise logical, comparison, memory dereference (load/store), reference (address-of argument, temporary or local label), and function call (both static and computed). Three additional operators are provided for manipulating the emulated stack: `Drop()`, `Dup(`$n = 0$`)` (which duplicates buried values when $n > 0$) and `Put(`n`)` (which stores into a buried location in the stack). These last two operations allow non-LIFO access to data on the stack.

Control flow is provided by unconditional `Br(n)` and conditional branches, `Bt(n)` and `Bf(n)`. The interpretation of 'truth' is the same as in C: zero is 'false', anything else is 'true'. (The VPU treats the condition codes register as an implicit value on the stack that is reified *only* when a logical result is used as an integer quantity. For example, there is no reification of the result of a comparison when the next operation is a conditional branch.)

The argument 'n' in the above branch instructions refers to a local label. Labels are maintained on a parallel stack, with *control scopes* being pushed and popped in a strictly LIFO fashion. Local labels are created and defined via three instructions:

- `Begin(n)` pushes `n` undefined local labels onto the label stack;
- `Define(n)` associates the current program counter value with the `n`th entry in the label on the stack; and
- `End(n)` pops the topmost `n` entries off the label stack.

Control is permitted to jump forward out of a local scope, implicitly truncating the emulation stack at the destination label. Conversely, a balanced stack is enforced for backward branches (loops). A local label can remain undefined providing it is never referred to within the function body. Attempting to jump to a label that is never `Define`d will raise an error at (dynamic) compile time.

Formal arguments are declared immediately after the prologue. The type of the argument is explicit in the declaring instruction (`Iarg()` or `Darg()`). Arguments are referred to within the function by position relative to the first (corresponding to the 'leftmost' argument in the equivalent C declaration); 0 is the first actual argument.A single `LdArg(`n`)` is provided to push an actual argument onto the stack; the type of the value pushed onto the stack is implicit and corresponds to the explicit type given in the instruction that declared the argument.

Temporaries are similar to arguments, but can be declared and destroyed at any point within the body of the function. For example, `Itmp` creates a new local `int` variable on the temporary stack. Subsequent `LdTmp(`n`)` and `StTmp(`n`)` instructions read and write temporaries, with $n = 0$ referring to the topmost (most recently declared) temporary. The `DropTmp(`n`)` instruction pops the topmost n temporaries off the stack.

```
int main(int argc, char **argv)
{
  VPU *vpu= new VPU();
  Label fn;
  vpu->Define(fn)->Ienter()->Iarg()
    ->Itmp()
    ->Ld(0)->StTmp(0)->Drop()
    ->Begin(1)->Define(0)
      ->LdTmp(0)->Ld("%d\n")
        ->Icall(2, (void *)printf)->Drop()
      ->LdTmp(0)->Ld(atoi(argv[1]))->Add()
        ->StTmp(0)
        ->LdArg(0)->Le()->Bt(0)
    ->End(1)
    ->DropTmp()
    ->Ld(0)->Ret()
    ->compile();
  fn(atoi(argv[2]));
  free(fn);
  return 0;
}
```

Figure 6: Dynamically constructing a function that uses local variables, labels and conditional branches. The program takes two command-line arguments: a loop 'step' and 'limit'. (The program compiles the first into the dynamic code as a constant and passes the second to it as an argument.) A temporary variable is created and initialised to 0. A local label is then created and defined. The value of the temporary is then printed, it is stepped and compared to the limit ('Le' is a 'less-or-equal' comparison); if the limit has not been reached the loop continues. The local label is then popped of the label stack and the local variable destroyed before returning 0. Running this program with the arguments '3 10' prints '0 3 6 9' on stdout. The indentation reflects the depth of the label and emulation stacks within the function body. Note that the Define instruction is overloaded to accept integer arguments (local labels on the label stack) and Label objects (global labels whose values remain available to the client application after compilation). Label is a convenience class provided by the VPU that includes overloaded definitions of operator() to simplify the invocation of dynamically-compiled code (eliminating the need to cast its address to a pointer-to-function type).

The compile() method requires that both the emulation and temporary stacks be empty after the final Ret() instruction in the function.

Figure 6 shows a rather more complete example that uses temporary variables, local labels and conditional branches.

3 Implementation of the VPU

Figure 7 shows the four phases of compilation within the VPU. The function described by the client is converted into an internal abstract representation.

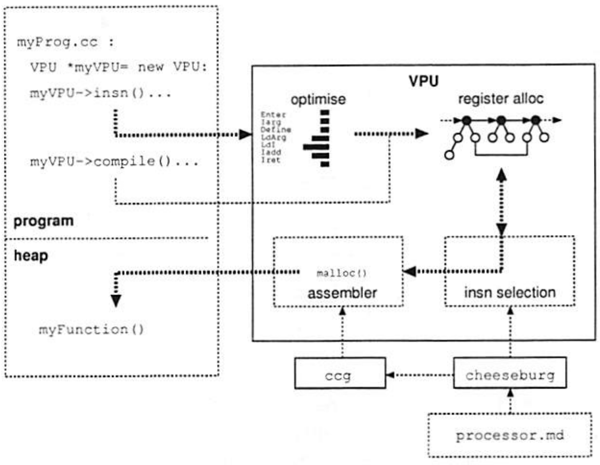


Figure 7: The VPU's architecture. An internal representation of the function and emulation stack is constructed in response to the client invoking methods on a vpu. Compilation involves performing analyses and optimisations on the internal representation, followed by instruction selection and register allocation (each of which can affect the other) to associate concrete instructions, physical registers and runtime stack locations with each abstract instruction. After sizing the final code space is allocated by calling malloc() into which a runtime assembler generates executable native code. Target-specific parts of the instruction selection and assembly phases are generated automatically from a processor description file by the program cheeseburg, similar in spirit to the iburg and lburg family of code generator generators.

Various optimisations are performed followed by concrete instruction selection and register allocation. Native code is then generated in memory, with a little help from ccg.

3.1 Creation and analysis of abstract code

This phase has the following goals:

- create of an abstract representation of the input function;
- verify the internal consistency of the function;
- perform architecture-neutral optimisations on the representation;
- resolve ambiguities in the stack arising from nonlinear control flow;
- eliminate dead instructions (unreachable or which compute unused values);

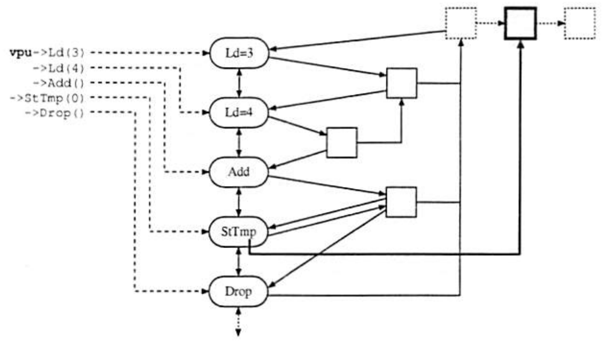


Figure 8: Internal representation of the abstract instructions and their input and output stacks. In this example the stack initially contains just temporaries and arguments. The two `Ld` instructions push new locations to the head of the stack; the tails of their respective output stacks point to the first location of their input stacks. The `Add` instruction consumes two input stack elements and pushes a new location for the result; the tail of its output stack therefore points to the third location of its input stack. The `StTmp` instruction has no stack effect and hence its input and output stacks are identical (and refer to the same location). The `Drop` instruction pops the first element off the stack; its output stack is therefore just the tail of its input stack.

- create optimal conditions for the following register allocation phase.

3.1.1 The abstract representation

A function is represented as a doubly-linked list of abstract instructions. Each instruction contains pointers to its *input stack* and *output stack*. The stack representation is similar to that described in [7] and consists of a linked list of stack *locations*. The "tail" of the stack is shared between successive instructions (rather than recreating a complete list of stack locations for each instruction), as illustrated in Figure 8.

This representation not only reduces memory requirements but also guarantees properties that are critical to the VPU's compilation process:

- the lifetime of any value on the stack is explicit: it begins with the instruction that creates the corresponding location and ends with the instruction that removes the location from its output stack;
- any two elements on the stack that represent the same logical value will also share identity (a single stack location object represents the value for all instructions that might reference it): a single location object represents a given value for the entirety of its lifetime.

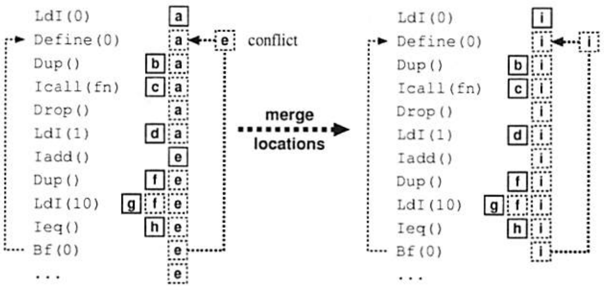


Figure 9: Contradictions in the stack caused by control flow. When two or more flows of control merge the VPU ensures that stack location identity is preserved between the branch and the destination. Whenever conflicts are detected (in this case because a loop iteration variable is kept on the stack) a new location is allocated and replaces the two original conflicting locations.

3.1.2 Control flow analysis

A control flow graph is constructed for the abstract representation. This graph is a set of $\{source \rightarrow destination\}$ tuples formed from the union of explicit control flow and that implied by linear flow into a label (represented by a `Define` instruction):

$$\{branch \rightarrow label\} \cup \{insn_{i-1} \rightarrow label_i\}$$

The graph is used primarily to detect and correct location ambiguities introduced by loops. This situation occurs, for example, when a loop uses a temporary location on top of the stack as an iteration variable. The output stack (at the point of the backwards branch) will not represent the same set of locations as the input stack (at the head of the loop). Figure 9 illustrates the problem. The VPU resolves this situation by replacing the two "colliding" locations with a newly-allocated location, which restores location identity over the lifetime of the loop.

3.1.3 Type analysis

Simple analysis is performed on the program to determine the input types of each instruction. This is necessary, for example, to unambiguously associate a concrete arithmetic instruction with a virtual instruction that has only one form (which might represent either an integer or floating point operation) and also to ensure that instructions having restricted types (shift instructions, memory references, etc., which are only defined for integer operands) are being used correctly.

For each instruction the types of the input arguments (if any) are verified to ensure that they are consistent with each other and with the output type of the instruction; if the output type is not known then it is inferred from the input types. The output type is stored in the instruction's output location for use in checking the input types of subsequent instructions.

The type checking phase also determines the "class" (void, integer, float, condition code, etc.) of the instruction's result. This class is stored in the output location for use during instruction selection. The combination of input classes and output class for a given instruction will be referred to below as the *mode* of the instruction.

3.1.4 Optimisations on the abstract representation

The VPU performs relatively few optimisations on the abstract program. The goal is to generate high-quality (but not necessarily optimal) code as quickly as possible. For a particular optimisation to be considered it must satisfy the following conditions:

- The VPU must be able to detect the opportunity for, and then implement, each optimisation *in parallel with* some other *essential* operation that would be necessary even for "unoptimised" compilation. In other words, all optimisations must be "piggy-backed" onto some other, required, traversal or manipulation of the abstract representation.

- Only optimisations that have a significant effect-to-cost ratio are considered.

- Global optimisations are not considered. (Their cost is always significant, requiring additional traversals of the abstract representation and/or the creation and maintenance of additional data structures.)

- Peephole optimisations, that transform particular sequences of instructions into a more efficient sequence, are also not considered. The costs associated with "pattern matching", and the need for an additional pass over the code, are not justified by the relatively small resulting gain in code quality [5].

Optimisations that do meet these criteria are constant folding, jump chain elimination and the removal of dead code. They can be performed (for example) in parallel with control flow or type analysis.

Constant folding is trivial. For a given instruction of arity n, if the topmost n elements of its input stack are never written and are associated with `Ld` instructions then:

- the n `Ld` instructions are deleted from the program;

- the constant result r of the operation is calculated;

- the original operation is transformed into `Ld(`r`)`.

Dead code occurs when a basic block is unreachable or when a side effect-free sequence of instructions computes a result that is never used. Any instruction that has a side effect (such as a store) sets an attribute s on its input and output location. For other operations of arity n, s is propagated from the output location to the n topmost locations on its input stack. During some *subsequent* traversal of the program, if s is unset for the input location of a `Drop` then both the `Drop` instruction and the instruction that generated the dropped location can be deleted from the program.[3] (Since locations are shared between all instructions that refer to them, a `Drop` occurring along one control path will never delete an instruction generating a value that is used along an alternate path through the program.)

Elimination of unreachable code consists of finding the transitive closure of reachable blocks starting from the entry point of the function. Any block not marked as reachable can safely be deleted. The algorithm is trivial and can be performed in parallel with the construction of the flow graph.

Jump chain elimination is performed in parallel with the construction of the control flow graph. The transformations are as follows:

$Br^+ \to Bx(L) \Rightarrow Bx(L)$ pulls any destination branch forward into a referent unconditional branch; and

[3]This is best done during a backwards traversal of the program, for example while assigning fixed and "constrained" registers to instruction output locations as described below.

$Bx \rightarrow Br^+(L) \Rightarrow Bx(L)$ pulls an unconditional branch forward into any referent branch.

These transformations are applied repeatedly for each branch instruction in order to find the destination of the chain.

3.1.5 Summary

At this point in the compilation we have:

- a linear program consisting of abstract instructions;
- an input stack and output stack attached to each instruction;
- a mode (type) associated with each instruction;
- a location in the emulation stack associated with each value used in the program (but not yet associated with any physical machine location);
- certain guaranteed conditions that simplify the subsequent phases of compilation, most importantly: no conflicts (contradictions in location identity) between the input stack at each label definition and the output stacks at each instruction that feeds control into the label.

3.2 Allocation of physical resources

This phase has the following goals:

- associate each sequence of one or more abstract instructions with a sequence of zero or more concrete machine instructions;
- determine the architectural characteristics and constraints that might affect register allocation (e.g, incoming/outgoing argument locations or register selections imposed by particular machine instructions);
- allocate machine resources (registers, physical stack locations) to each reified value in the emulation stack while: respecting architectural constraints, avoiding move chains and minimising the number of concrete instructions generated for each abstract instruction.

3.2.1 Instruction selection

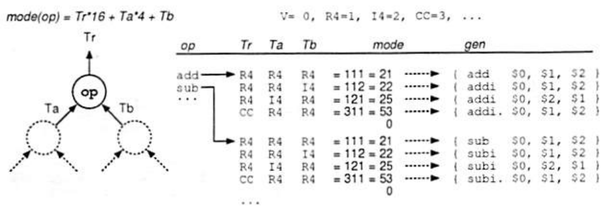
Instruction selection determines which concrete machine instructions should be emitted to implement a given abstract instruction in the program.

Instruction selection and register allocation are intimately related. The selection of instructions determines when (and possibly which) registers should be allocated. Register allocation in turn can cause spill and reload code to be inserted into the program which in turn will use registers (which have to be allocated). In any case, register selection cannot begin until an initial instruction selection has determined:

- whether a given combination of operation and input/output modes is supported directly by the hardware or whether a sequence of machine instructions is required to synthesise the required operation;
- whether or not a given literal value can appear as an immediate operand (according to its size and whether the hardware supports an immediate in the corresponding operand position);
- whether an operation must reify an integer value in a register (for example, returning a logical value as the result of a function call).

Several approaches to instruction selection are possible. The simplest is to implement an exhaustive set of instruction emitters that covers all possible combinations of *operation* × *operand* mode(s). This approach has severe drawbacks:

- the number of emitters undergoes combinatorial explosion (from the number of possible permutations of operand modes);
- exhaustive case analysis is required to determine the correct emitter for a given combination of operation and operand(s) (`switch`es inside `switch`es inside...);
- the case analysis code is difficult (or even impossible) to generate automatically. It must be written (and maintained) manually;
- the resulting code generator is relative simple, but large (because of a high degree of repetition) and slow (because of the many conditional branches and indirect jumps in the case analysis).



```
V= 0, R4=1, I4=2, CC=3, ...

op      Tr  Ta  Tb   mode             gen
add --> R4  R4  R4   = 111 = 21  ---> { add   $0, $1, $2 }
sub     R4  R4  I4   = 112 = 22  ---> { addi  $0, $1, $2 }
...     R4  I4  R4   = 121 = 25  ---> { addi  $0, $2, $1 }
        CC  R4  R4   = 311 = 53  ---> { addi. $0, $1, $2 }
                             0

        R4  R4  R4   = 111 = 21  ---> { sub   $0, $1, $2 }
        R4  R4  I4   = 112 = 22  ---> { subi  $0, $1, $2 }
        R4  I4  R4   = 121 = 25  ---> { subi  $0, $2, $1 }
        CC  R4  R4   = 311 = 53  ---> { subi. $0, $1, $2 }
                             0
        ...
```

Figure 10: Table-driven instruction selection in the VPU. The output and input operand mode(s) are encoded as a numeric signature. A table associated with each operation maps mode signatures onto emitter functions each of which deals with one particular combination of modes. (In this diagram the function pointers are elided and instead the assembler template within the emitter is show in its place.)

```
RI4: Add(RI4, LI4) { #[ addi r($0), r($1), $2     ]# }
RI4: Add(RI4, RI4) { #[ add  r($0), r($1), r($2) ]# }
CCR: Cmp(RI4, LI4) { #[ cmpi r($1), $2            ]# }
RI4: Cmp(RI4, LI4) { #[ cmpi r($1), $2            ]#
                        setcc($0, insn->op)          }
```

Figure 11: Part of the processor description for PowerPC. Each line describes the instruction(s) to generate for a given combination of input and output modes (such as RI4, representing to a 4-byte integer register). The assembler between '#[' and ']#' delimiters is converted into code that generates the corresponding binary instructions (by the ccg preprocessor, see Section 3.3). The positional arguments refer to the modes in each "rule" and are replaced with expressions representing the physical location (register, stack offset, constant) corresponding to that mode. (setcc is a function that emits code to place 0 or 1 in a register depending on the state of the condition codes and a given logical relation.)

At the other extreme is the iburg approach which transforms a formal, bottom-up rewrite grammar into a program that finds minimal cost covers of arbitrary subtrees in the intermediate representation. Each cover represents a (sequence of) machine instruction(s) to be emitted. In general, by maximising the number of tree nodes consumed by each cover the code generator minimises the number of instructions generated. This approach suffers from search complexity and the need for backtracking in the algorithm, both of which slow down code generation noticeably and progressively as more highly-optimised covers are added to the grammar to deal with obscure cases. (This is especially significant when all other phases of compilation are designed to minimise compilation time.)

The VPU takes an intermediate approach. It uses a table-driven, non-backtracking algorithm and a few simple heuristics to determine an optimal instruction sequence for a given combination of operation and input/output modes (in effect, a minimal cost cover for a "subtree" of depth one in the intermediate representation).

Figure 10 illustrates the table used to select machine instructions for an abstract instruction. (The nodes of the 'tree' in the VPU's abstract program are the locations in the simulation stack rather than the operations themselves.) These tables are generated trivially from a processor description file, part of which is shown in Figure 11.

For a given operation, the input and output mode(s) are combined into a numeric signature. Instruction selection searches the table associated with the operation to find an emitter function matching the signature (which it stores in the instruction for use during final assembly). If no emitter is found (which means the mode is illegal) then the first input operand that is neither a register nor a constant is forced into a register and the search repeated. If no match is found with only register and literal inputs then the first non-register operand is converted to a register and the search repeats. If there is still no match when all operands are in registers then the table (which must provide register-register modes for all instructions) is necessarily incomplete, indicating an error in the machine description file itself.

This algorithm is much faster than BURG-style instruction selection and yet results in a similar quality of generated code on RISC processors.

After instruction selection we know precisely:

- the final class (constant, integer/floating point register, void, etc.) of each location in the emulation stack (and hence the required mode for every abstract operation);
- the locations for which machine registers must be allocated;
- the emitter function corresponding to each operation for its particular mode (cached in the instruction for use during final code generation, as described below).

3.2.2 Register allocation

Before final register allocation can begin, the code generator must satisfy any constraints imposed by the architecture on the choice of registers. Three kinds of constraints must be dealt with:

- input constraints: for example, on the Pentium we have no choice but to place dividends in register `eax`;
- output constraints: for example, on the Pentium we have no choice but to retrieve the quotient and remainder from the register pair `eax:edx`;
- volatile registers: for example, on the PowerPC registers `r3` through `r12` are clobbered across function calls.

(The need to pass outgoing arguments, and find incoming arguments, in particular registers is just a particular combination of the above constraints.)

A separate pass is made through the program to preallocate constrained registers in their associated emulation stack locations. The bulk of the algorithm is as follows. For each instruction, in *reverse* order (from the last to the first):

- if the input is required in a particular register and this register is not flagged as clobbered in the instruction, then
  - assign the register to the instruction's output location
- if the instruction clobbers one or more registers, then
  - iterate over the instruction's output stack adding the register(s) to the set of clobbered registers for each emulation stack location (final register allocation will avoid allocating a register to a given location if it is marked clobbered in that location); and
  - remove any preallocated register for the location if it coincides with one the clobbered register(s).

Final register allocation can now be performed. The allocator creates a bit mask for each register class (integer, float, etc.) representing the set of available registers in that class and then iterates (forwards) over the program. For each instruction:

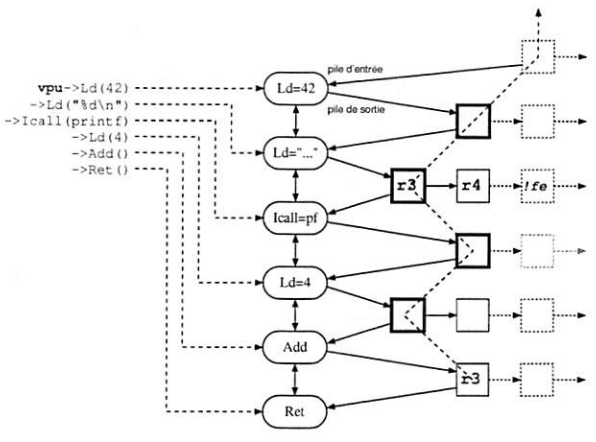


Figure 12: Allocation of constrained registers. An initial backwards pass is made through the program. The final `Ret` instruction requires its input in `r3` (on the PowerPC). The earlier call instruction requires its two arguments in `r3` and `r4` and also clobbers registers `5` through `10` in all locations beneath them on the stack (represented here by the mask '`!fe`').

- if the instruction consumes inputs then add any registers associated with its input locations to the appropriate mask;
- if the instruction generates a value in a register and the output location has not yet been allocated a register, then remove a register from the appropriate mask and assign it to the output location;
- if the instruction occurs at a basic block boundary (branch, label definition or call) then rebuild the register masks by
  - resetting them to their initial state and
  - iterating over the instruction's output stack, removing all registers encountered from the mask.

This process is illustrated in Figures 12 through 14.

3.2.3 Register spill and reload

If the register allocator runs out of available registers then it must choose a register to free up for allocation, by spilling it into the stack and then reloading it later (sometime before the instruction that uses its value). It is difficult to determine the optimal choice of register to spill without employing expensive algorithms, however a good choice (and frequently optimal) is to spill the register that is

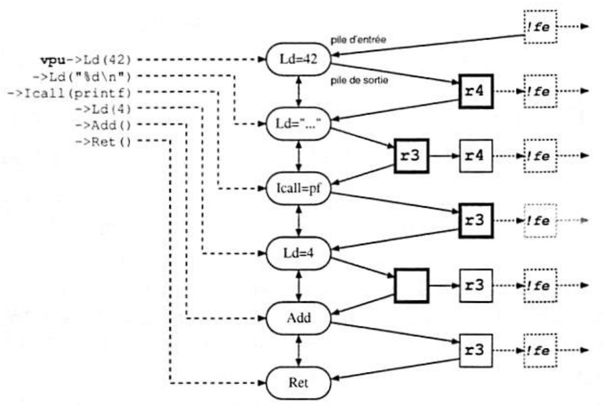


Figure 13: The situation before final allocation begins. Since locations are shared, any registers constraints and clobbered register sets are instantaneously propagated forwards and backwards to all instructions that might be affected.

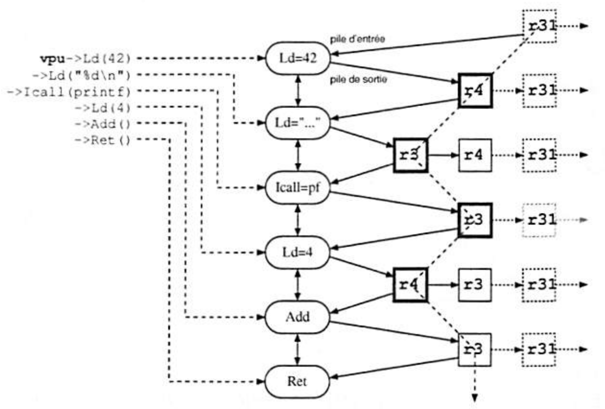


Figure 14: Final register allocation. A forward pass is made through the program to complete register allocation. Registers allocation respects the clobbered register masks stored in each location. The presence of the call instruction sets this mask to `!fe` for the lowest (shown) location in the stack thereby preventing a call-clobbered register being allocated to it; instead it is allocated the call-saved register `r31`.

most distantly used (MDU). An excellent approximation to the MDU register is available immediately to the allocator in the emulation stack attached to each instruction. The deepest location containing a register of the required class typically corresponds to the MDU. The allocator therefore scans the stack (below the location for which it is trying to allocate) to find this register and then inserts code just before the current instruction to spill it into a stack location. This is illustrated in Figure 15.

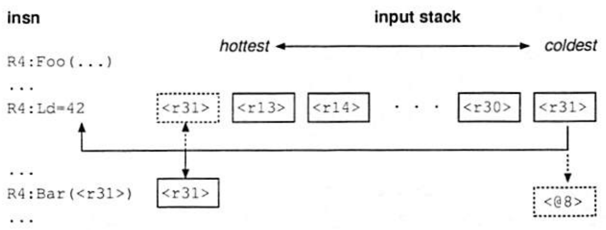


Figure 15: Spilling a register. The instruction `Bar` requires its input (a constant) in a register but none are available. The allocator scans the input stack to find the deepest register of the required class, in this case `r31`. The corresponding location is changed to refer to a location in the runtime stack (frame offset `8` in this example) and the register reused.

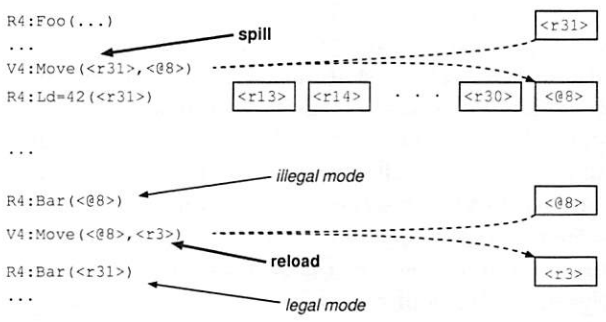


Figure 16: Reloading a register. The spilled location no longer represents a legal mode for its subsequent use in the `Bar` instruction. The instruction selection algorithm reconverts the location into a register (such that `Bar`'s mode is legal), reallocates a register to hold the reloaded value and inserts a `Move` pseudo-instruction into the program to effect the reload.

Register reload is performed implicitly by rerunning instruction selection for instructions that consume spilled locations. If the new mode is legal then the instruction can consume the value directly from the spilled location and no further action is needed. Otherwise the selection algorithm will convert the spilled input argument(s) to register modes (as described earlier), insert a 'move' pseudo-instruction into the program to mark the reload, and reallocate registers for the newly-inserted instruction. Figure 16 illustrates this process.

3.3 Final assembly

All that remains is to iterate over the code and call the emitter function attached to each abstract instruction in the program. (The emitter function ap-

CPU	*OS*	*libvpu.a*	*emit.o*
PowerPC	Darwin	158,725	52,212
PowerPC	GNU/Linux	175,612	52,088
Intel 386	GNU/Linux	124,791	31,374

Table 1: Compiled size of the VPU for 10,600 lines of C++ source code. Approximately one third of the compiled size, but less than 5% of the source code, is accounted for by the instruction emitters (inlined calls to `ccg` macros). With a little work the size of these emitters could be reduced significantly (by replacing the inline-expanded macros with function calls) at the cost of slightly slower final instruction assembly.

propriate for a given operation and operand modes is found during instruction selection and stored in the abstract instruction to avoid having to rescan the tables.) The code generator performs this iteration twice. The first iteration generates code but does not write any instructions into memory. Instead the program counter (PC) is initialised to zero and the emitter called for each instruction in turn; for each `Define` instruction, the value of the PC is stored in the associated local label. At the end of this first iteration the PC will contain the size of the final code and each local label will contain the offset of the label relative to the first instruction in the generated code. Memory is then allocated for the final code (by calling `malloc()` or some other, application-defined memory allocator) at some address M. Each local label then has M added to its (relative) value to convert it to its final, absolute address. Finally, the PC is set to M and a second iteration made over the program to generate binary instructions in memory at their final locations.

The assembler templates for binary instructions are written using a runtime assembler generator called `ccg`. A full description of it is beyond the scope of this paper, but it should be noted that the cost of assembling binary instructions using `ccg` is very low: within the emitter functions, an average of 3.5 instructions are executed for each instruction generated in memory.

4 Evaluation

At least five metrics are important when evaluating a dynamic code generator: the size of the code generator itself, its compilation speed, the memory requirements during compilation, the size of the generated native code and the execution speed of that code.

CPU	*clock*	*v-insns/sec*	*binary/sec*
PowerPC G3	400 MHz	288,400	1.1 MB
PowerPC G4	1 GHz	610,000	2.5 MB
Intel P3	1 GHz	656,108	1.8 MB
Intel P4	3.6 GHz	1,611,111	4.1 MB

Table 2: Compilation speed. The third column shows the number of virtual instructions compiled per second, and the final column the amount of native code generated per second. These figures were calculated by compiling 20 different input functions (of between 7 and 80 virtual instructions) 10,000 times in a loop. A total of 4,350,000 virtual instructions were compiled into 17.5 MBytes (PowerPC) or 11.7 MBytes (Intel) of native code, taking between 15 seconds (on the slowest machine) and 2.7 seconds (on the fastest).

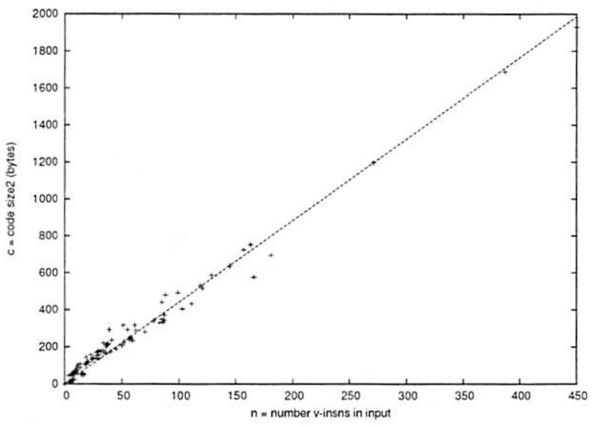


Figure 17: Compiled native code size c (bytes) per n virtual instructions, for each function of a medium-sized program (a Lisp-like dynamic language and its runtime system). The dotted line is a linear fit of the data points: $c = 4.4n$.

The VPU is a little over 10,600 lines of C++ source code. Table 1 shows the size of the compiled library (without symbols): about 170 KBytes and 125 KBytes on on PowerPC and Intel architectures, respectively.

Table 2 shows the compilation speed, averaged over 16 different functions of various sizes, compiled (and then `free()`ed) 10,000 times in a loop. On three-year-old PowerPC hardware the VPU compiles 288,000 virtual instructions per second (generating a little over 1 MByte of code), and about 656,000 instructions per second (for a little over 1,8 MByte of native code) on Intel Pentium. On current hardware the figures are 610,000 instruc-

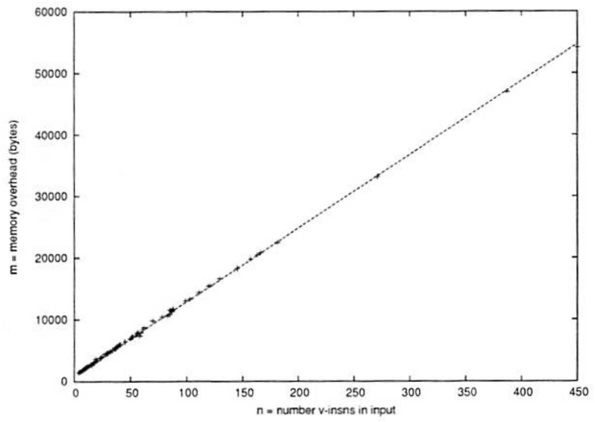


Figure 18: Memory overhead m (bytes) per n virtual instructions in each function of a medium-sized program. The dotted line is a linear fit of the data points: $m = 908 + 120n$.

tions/second (2.5 MBytes of native code) and 1.6 million instructions/second (over 4 MBytes of native code) on PowerPC and Pentium, respectively.

Figure 17 shows the generated code size plotted as a function of the number of virtual instructions per function for a much larger program (a dynamic compiler and runtime support for a complete, Lisp-like programming language). The native code size is approximately 4.4 bytes per virtual instruction on the PowerPC.

Figure 18 shows the memory requirements during compilation (for the same Lisp-like language and runtime system). Instantiating a VPU costs a little under 1 KByte of memory, with an additional 120 bytes required per virtual instruction added to that function. (All of this memory, other than that required to hold the native code, is released by the VPU once code generation is complete.)

The code produced by the VPU is typically between 10% and 20% larger than than that produced by '`gcc -O2`' for an equivalent program. Numerical benchmarks run at between 90% and 115% the speed of the equivalent programs compiled with '`gcc -O2`'.

The VPU has been used to implement many different kinds of language runtime support; for example, dynamic binding (for dispatch to virtual functions) with an inline cache. Dispatching through a VPU-generated inline cache costs approximately 1.66 times a statically-compiled C function call.

5 Conclusions

The VPU is a plug-and-play dynamic code generator that provides application support for runtime generation of C ABI-compatible native code. Integrating the VPU into any application is trivial, after which it can be used to generate arbitrary functions (from simple "partial evaluation" type optimisations through compiling scripting languages into native code for better performance). It is also an ideal component for the core of portable "just-in-time" compilers for dynamic languages, where a VPU-based dynamic code generator can be added with very little work.

Several projects are currently using the VPU aggressively. It is the execution engine for the YNVM [10], a dynamic, interactive, incremental compiler for a language with C-like semantics, a Lisp-like syntax (in other words, a C compiler in which programs and data are the same thing) and the same performance as statically-compiled, optimised C. The JNJVM [11] is a highly-reflexive Java virtual machine built entirely within the YNVM that uses the VPU directly to generate code for Java methods. Outside the domain of languages the YNVM (with a VPU inside) has been used to create C/SPAN [8], a self-modifying web cache that modifies its cache policies (by dynamically recompiling new ones) in response to fluctuations in web traffic and network conditions.

References

[1] `http://www.cminusminus.org`

[2] `http://www.cs.virginia.edu/zephyr/papers.html`

[3] `http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/INTRO.html`

[4] `http://research.sun.com/research/self/papers/papers.html`

[5] M. Chen, K. Olukotun, *Targeting Dynamic Compilation for Embedded Environments.* 2nd USENIX Java Virtual Machine Research and Technology Symposium (Java VM'02), San Francisco, California, August 2002, pp. 151–164.

[6] D.R. Engler, *VCODE: a Retargetable, Extensible, Very Fast Dynamic Code Generation System.* SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1996, pp. 160–170.

[7] A. Krall, *Efficient JavaVM Just-in-Time Compilation.* International Conference on Parallel Architectures and Compilation Techniques, Paris, 1998, pp. 205-212.

[8] F. Ogel, S. Patarin, I. Piumarta and B. Folliot, *C/SPAN: a Self-Adapting Web Proxy Cache.* Workshop on Distributed Auto-adaptive and Reconfigurable Systems, ICDCS 2003, Providence, Rhode Island, May 2003.

[9] Ian Piumarta, *CCG: A Tool For Writing Dynamic Code Generators.* OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, Denver Co, November 1999.

[10] I. Piumarta, *YNVM: dynamic compilation in support of software evolution.* OOPSLA'01 Workshop on Engineering Complex Object Oriented System for Evolution, Tampa Bay, Florida, October 2001.

[11] G. Thomas, F. Ogel, I. Piumarta and B. Folliot, *Dynamic Construction of Flexible Virtual Machines,* submitted to Interpreters, Virtual Machines and Emulators (IVME '03), San Diego, California, June 2003.

[12] `http://www-sor.inria.fr/projects/vvm/realizations/ccg/ccg.html`

[13] `http://www.gnu.org/software/lightning/lightning.html`

LIL: An Architecture-Neutral Language for Virtual-Machine Stubs

Neal Glew Spyridon Triantafyllis* Michał Cierniak†
Marsha Eng Brian Lewis James Stichnoth

Microprocessor Technology Lab, Intel Corporation

Abstract

High-performance MREs (managed runtime environments) that run either Java[1] or CLI applications require machine code sequences, called *stubs*, to implement such runtime support operations as object allocation, synchronization, and native method invocation. Due to the frequency of these operations, implementing stubs efficiently is critical for performance. Also, the number of different stubs that have to be created and maintained makes stub creation a sizable part of an MRE's implementation. Stubs typically require access to low-level resources such as registers and the call stack, and often must be specialized at runtime for particular classes or methods. Although stubs can be implemented by generating hand-crafted machine code at runtime, this approach is tedious and error-prone, and leads to stubs that are non-portable and difficult to maintain.

To address these problems, we designed a domain-specific language, called LIL, for implementing stubs. LIL is low-level but architecture-neutral, allowing the creation of stubs that are both portable and efficient. LIL also abstracts away many implementation details, making stubs easier to read. It is lightweight enough to be used for dynamic stub generation. LIL's validity checker helps us to catch many errors early. Our preliminary experience using LIL indicates that it greatly eases development and maintenance of stubs without sacrificing performance.

1 Introduction

The *Open Runtime Platform* (ORP [5]) is an experimental managed runtime environment (MRE) developed by the Programming Systems Lab at Intel. ORP supports both Java [11] and Common Language Infrastructure (CLI [8]) applications. To provide a credible starting point for experimentation, ORP's performance is comparable to that of other state-of-the-art MREs. Also, ORP supports multiple platforms with as little effort as possible. To date, ORP runs on both Intel® IA-32 and Itanium® Processor Family (IPF) architectures, and on both the Windows and Linux operating systems.

One of the most distinctive features of ORP is its modularity. ORP comprises three components: a just-in-time compiler (JIT), a garbage collector, and a core virtual machine (VM). Interaction between these components is allowed only through strictly defined interfaces. Thus the VM, the JIT, and the garbage collector are isolated from each other's implementation details. This greatly simplifies experimentation. For example, we can experiment with new compilation techniques or garbage collection approaches by modifying just the JIT or the garbage collector respectively, without having to modify other ORP components. To date, we have implemented seven different JITs (see, *e.g.*, [2, 7, 4, 1]) and five different garbage collectors for use within ORP.

ORP uses optimized machine code sequences, called *stubs*, to implement a number of common runtime support operations including object allocation, synchronization, and exception handling. Due to ORP's modular design, the JIT does not possess enough information to directly generate the code for most of these stubs. Instead, the JIT has to rely on support from the VM, and sometimes from the garbage collector as well. These runtime support operations usually require direct access to low-level machine resources such as registers and the call stack, and often need to be generated dynamically at runtime.

In the past, the ORP VM generated stubs by directly emitting hand-written machine code. However, this approach is error prone and results in stubs that are difficult to maintain. Stubs generated in this way are also not portable: a different

*Spyridon is currently at Princeton University.
†Michał is currently at Microsoft Corporation.
[1]Other brands and names are the property of their respective owners.

version of each stub is needed for each of the platforms supported by ORP. This situation led us to design LIL, a domain-specific language for expressing runtime support functionality. In this paper, we present LIL and argue that it greatly simplifies the creation of stubs, both by making stubs more concise and readable, and by hiding architecture-dependent details. Furthermore, LIL provides these benefits without performance loss.[2] Indeed, using LIL facilitates experimentation with stub code and with ORP's runtime support system in general, possibly leading to performance improvements.

In the next section, we describe stubs in more detail, motivate the LIL approach, and present an object allocation stub as an example that illustrates some of the issues involved. Sections 3 and 4 present the LIL language in detail and outline its benefits. Section 5 presents preliminary results showing that LIL's performance is similar to hand-written assembly stubs. Finally, sections 6 and 7 discuss related and future work.

2 Stubs

2.1 Motivation of Our Approach

Runtime support stubs are ubiquitous in ORP. Examples include object allocation, type checking (`instanceof`, `checkcast`, and `aastore` JVM bytecodes), exception throwing, native-method invocation (JNI), delayed method compilation, synchronization, arithmetic helpers, and class initialization. Since these operations appear often in Java applications, implementing stubs efficiently is critical for performance. Also, the sheer number of different stubs that must be supported by ORP makes it necessary to generate stubs in a portable and maintainable way. The purpose of LIL is to answer both these challenges.

Several approaches have been used in the past to implement runtime support code:

1. Stubs can be written in a high-level language such as C, and compiled with the rest of the VM source code. The VM then responds to requests for runtime support routines by passing the addresses of these precompiled functions to the JIT. In the past, ORP used this approach for a few stubs.

2. Stubs can also be implemented in assembly language.

3. The VM can generate machine-code stubs at runtime. This was ORP's approach before LIL.

4. The VM can express runtime support stubs in Java bytecode. The JIT then compiles these stubs in the usual way. Operations that cannot be expressed in bytecode, such as those that violate Java's type system, can be handled through "magic" method calls that are recognized and inlined into machine code by each JIT. This approach is taken by the Jikes RVM [3], and is discussed further in Section 6.

5. The VM can generate stubs directly in the JIT's IR.

None of these existing approaches fit ORP's needs, for the following reasons.

1. **Dynamic code generation.** While some stubs can be generated statically, at ORP build time, others must be customized at runtime. For example, the stubs for invoking native methods must be generated for each native method. The set of native methods is not known until class-load time, and changes as new classes are loaded. This dynamic code generation precludes using precompiled stubs, as in the first and second approaches above.

   In addition, many frequently executed operations, such as type checking and object allocation, are significantly more efficient if they are customized on a per-type basis. For example, the ORP VM can generate a custom `checkcast` sequence for each class in a program, which is significantly more efficient than a general implementation of `checkcast`.

2. **Low-level access.** Many runtime support operations need to directly access low-level resources such as processor registers or the call stack. For example, ORP's IPF implementation keeps some global values in registers, including the pointer to the data structure for the current thread. Examples of operations that need to directly manipulate the stack include exception throwing, object allocation, and native method invocation. Such low-level operations cannot be implemented in a high-level language, Java bytecode, or most JIT IRs.

   Furthermore, certain frequently invoked stubs are so performance-critical that they must take full advantage of the underlying processor. This makes it necessary to implement such stubs in hand-crafted machine code, or at least in a sufficiently low-level language.

[2]Actually we still need a few hand-written stubs to get the best performance. We expect that future tuning of LIL will eliminate this need.

The figure below uses the following constants: `fpo` is the offset of the frontier pointer in the per-thread structure and `lpo` is the offset of the limit pointer in the per-thread structure.

	IA-32	IPF	LIL

1.	`push ebx`
	`push ebp`
	`mov ebx, [esp+20]`
	`mov ebp, [esp+16]`
			`entry 0:managed:`
			`  g4,pint:ref;`
			`locals 3;`

2.	`mov ecx, fs:[0x14]`	`adds r18 = 0h, r4`	`l0=ts;`

3.	`mov eax, [ecx+fpo]`
	`mov edx, [ecx+lpo]`
		`adds r14 = fpo, r18`
		`ld8 r8 = [r14]`
		`adds r15 = lpo, r18`
		`ld8 r16 = [r15]`
			`ld r,[l0+fpo:ref];`
			`ld l1,[l0+lpo:pint];`

4.	`add ebx, eax`
	`cmp ebx, edx`
	`ja slowpath`
		`add r17 = r8, r32`
		`cmp.ltu p0,p6=r16,r17`
			`l2=r:pint+i0;`
			`jc l2 >u l1,slowpath;`

5.	`mov [eax], ebp`
	`mov [ecx+fpo], esi`
	`pop ebp`
	`pop ebx`
	`ret 8`
		`(p6) st8 [r8] = r33`
		`(p6) st8 [r14] = r17`
		`(p6) br.ret b0`
			`st [r+0:pint],i1;`
			`st [l0+fpo:pint],l2;`
			`ret;`

6.	`slowpath:`
	`pop ebp`
	`pop ebx`
	`/* push_m2n */`
	`push [esp+36]`
	`push [esp+44]`
	`call gc_alloc`
	`add esp, 8`
	`/* pop_m2n */`
	`ret 8`
		`/* push_m2n */`
		`adds r53 = 0h, r32`
		`adds r54 = 0h, r33`
		`brl.call gc_alloc`
		`/* pop_m2n */`
		`br.ret b0`
			`:slowpath;`
			`push_m2n 0;`
			`in2out platform:ref;`
			`call gc_alloc;`
			`pop_m2n;`
			`ret;`

Figure 1: IA-32, IPF, and LIL code sequences for object allocation. The `push_m2n` and `pop_m2n` code sequences are omitted for brevity.

3. **Portability.** The requirement to generate low-level code suggests using the second and third approaches, that is, using assembly language to implement stubs. However, this causes portability and maintainability problems. New assembly-language implementations would be needed for each new processor and operating system. Since ORP contains many stubs, this would be onerous.

4. **VM/JIT dependencies.** Maintaining a clean interface between the JIT and the VM is a key ORP goal. Thus we cannot use the fifth approach, generating stubs directly in the JIT's IR, since this approach would tie the VM to a particular IR, and perhaps to a particular JIT implementation. It would be necessary to rewrite the stubs each time the JIT's IR is changed.

5. **VM/MRE dependencies.** Because ORP supports both Java and CLI, approach 4 requires that one of these languages be chosen as the stub implementation language, making ORP asymmetric and more dependent upon that language. An alternative would be to write all stubs in both languages, which is as onerous as writing them for multiple architectures.

6. **Stub inlining.** Since runtime support stubs are invoked often, implementing them using called functions can incur significant call overhead. Directly inlining runtime support stubs into JIT-generated code would reduce this overhead. Approaches 1–3 are clearly not suitable for this purpose. Although approaches 4 and 5 could achieve this, they suffer the limitations noted above. Section 4.5 discusses how LIL can be used to provide a solution to this problem.

For these reasons, none of the existing approaches for generating runtime support code is appropriate for ORP, which motivated us to develop LIL. In the rest of this paper we argue that the use of LIL provides an attractive balance between performance, portability, and modularity. The main drawback of LIL is that it requires another language and compiler within ORP. This drawback is mitigated by the simplicity of the language—the implementation adds only 8000 lines of code.

In the remainder of this section, we give an example, the object allocation stub, and show how it is implemented in IA-32 and IPF assembly code, and in LIL. This example illustrates some of the issues involved in creating stubs. Finally, before presenting the LIL language in Section 3, we discuss in more detail the thread-local storage and stack-marking issues, because they occur so often in stubs for managed runtime environments.

2.2 Example

As an example of runtime support code, consider the object allocation stub. Object allocation is implemented by the garbage collector. A high-performance garbage collector typically provides each thread with its own thread-local allocation area. This allows a thread to allocate new objects quickly, without having to acquire a global heap lock. The thread-local allocation area can be represented as a frontier pointer and a limit pointer. The allocation sequence can be divided into a fast path and a slow path. First, the requested object size (including all necessary alignment padding) is added to the frontier pointer, and the result is compared against the limit pointer. If enough space is available, the fast path is executed. This path updates the frontier pointer, initializes the newly allocated object by clearing all its fields and setting its virtual-method table (*vtable*) pointer, and returns it. If the available space is not enough, the slow path is executed. This path marks the execution stack in case garbage collection and associated stack walking are necessary, and calls into the garbage collector to perform the allocation.

Figure 1 shows the code for object allocation. Columns one and two show the hand-written IA-32 and IPF assembly code, and column three shows the equivalent LIL code, which is described in detail in Section 3. The stub is intended to be called directly from JIT-generated code, and is called with two arguments: the object size in bytes and the vtable pointer that corresponds to the object type. Notice the differences between the first two columns—the different instruction sets and the different calling conventions. The purpose of LIL is to abstract away these differences by generating platform-specific code sequences automatically. In this way, the VM could be ported to a new platform without needing to rewrite and maintain a large body of custom stub code.

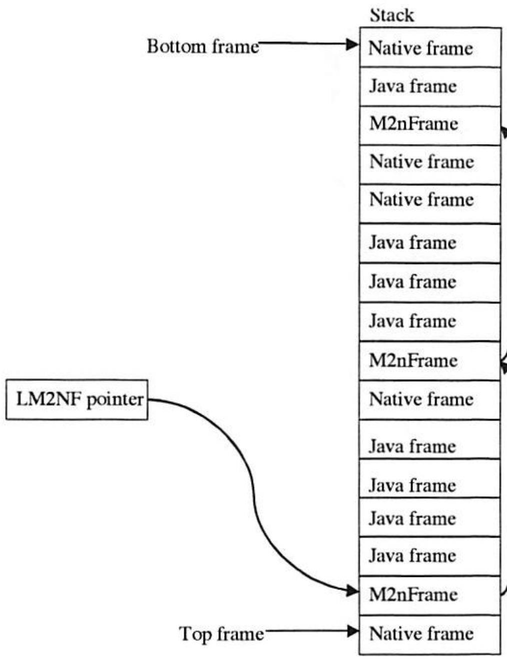


Figure 2: List of M2nFrames used for stack walking.

2.3 MRE-Specific Issues

Another difference between columns one and two of Figure 1 concerns MRE-specific details. ORP has a per-thread structure that holds information about each thread, including the frontier and limit pointers for thread-local memory allocation. A pointer to this structure is stored in the thread-local storage provided by the platform's threading system. Thus, a stub can obtain a pointer to the ORP thread structure for the current thread by loading it from the thread-local storage. We call this operation *loading the thread pointer*. How it is done varies by both architecture and operating system. Row two of Figure 1 shows how it is achieved on Win32 operating systems. On the IA-32 architecture, it is loaded from a fixed offset in the `fs` segment; on the IPF architecture, ORP keeps the pointer in register `r4` (normally a preserved register). On Linux, the pointer is obtained by calling a function provided by the `pthreads` library.

Another MRE-specific issue is `M2nFrames`. Each time JIT-generated code calls native code, ORP requires that a marker be inserted between the two activation frames on the stack. This marker is necessary because of the way ORP implements stack walking, used in tasks such as root-set enumeration, exception propagation, and stack introspection. We call these markers managed-to-native frames, or `M2nFrames`. Figure 2 shows the layout of a typical thread's stack. Combining information stored in the `M2nFrames` with the available information on managed code, the VM can easily locate each managed activation frame on the stack. The details of this scheme are not important to this paper. What is important is that any stub that might cause stack walking, either directly or indirectly, must push an `M2nFrame` onto the stack upon entry, and pop it before returning. The `M2nFrame` is actually part of the stub's activation frame. Also important to this paper is that the details of setting up `M2nFrames` are specific to the implementation of the VM and to the architecture. Therefore, stub code cannot be made portable unless these details are abstracted away.

3 LIL

We designed the LIL[3] language in order to enable the creation of platform-neutral stubs in ORP. We have implemented LIL, including code generators for the IA-32 and IPF architectures, within the ORP framework. This section introduces LIL by example and then provides a detailed description of the language.

3.1 Example

The last column of Figure 1 depicts a LIL stub for allocating an object. This stub is compiled into code that acts like a function. The stub's `entry` line states that it is called using the managed-code calling convention[4] with two arguments, and that it returns a result. The arguments are of type `g4` (32-bit general-purpose value) and `pint` (pointer-sized general-purpose value, often used for pointers other than object references), and the result is of type `ref` (reference to an object in the heap). The "0" is the number of standard places, which are described in Section 3.2.1. The rows in the figure do the following:

1. This row has the entry declaration and declares that the rest of the stub will use three local variables.

[3]LIL stands for Low-level Intermediate Language, and its pronunciation suggests its "little"-ness or lightweight nature.

[4]ORP specifies a calling convention that all JIT-compiled code must conform to; this is called the *managed-code calling convention*.

2. This row loads the thread pointer into `l0` (the first local variable).

3. This row loads the frontier and limit pointers. Both instructions load from the address equal to `l0` plus a constant (`fpo` and `lpo` respectively); these constants are the offsets of the frontier and limit pointers in the ORP thread structure. The first instruction loads a `ref` into `r` (the return variable); the second loads a `pint` into `l1`.

4. This row computes the new frontier pointer and compares it to the limit pointer. The first instruction adds `r` and `i0` (the first input, the size of the object to be allocated) and places the result in `l2`. The second instruction branches to label `slowpath` if `l2` is greater than `l1`, otherwise it continues with the next instruction.

5. This row initializes the object's vtable pointer, updates the frontier pointer, and returns. The first two instructions store `pints` at the addresses `r` and `l0` plus the constant `fpo` respectively. In the first instruction, the value stored is `i1` (the vtable pointer), in the second it is `l3`. The final instruction returns, and the value returned to the caller is the current value of `r`.

6. This row is the slow path. It starts with the declaration of label `slowpath`, the target of the conditional jump above. Next it pushes an `M2nFrame`. Then it sets up to call a function according to the platform calling convention (*i.e.*, the calling convention used by the platform's C compiler), using the arguments to the stub as the arguments for the call. Then it calls `gc_alloc` and sets `r` to the value returned. Finally it pops the `M2nFrame` and returns.

3.2 The Language

A LIL stub specifies code that is like a function in a high-level language. Like a function, it takes arguments and can return a result. The stub specifies which of several calling conventions it conforms to. Also like a function, each stub executes with an activation frame on the stack. Conceptually, this activation frame is divided into a number of areas that can vary in size and type across the execution of the stub. For our purposes, these areas are: inputs, standard places, locals, outputs, and return. The inputs initially hold the arguments passed by the caller, but they can change by assignment. Their number and type is fixed across the stub. Standard places are described in Section 3.2.1. The locals hold values local to the stub. Their number is determined by `locals` declarations, and their types are determined by a simple flow analysis. The outputs hold values passed to functions called by the stub. Their number and types are determined by `in2out` and `out` declarations. These declarations set up an output area, and various call instructions perform the actual call. In the case of `out` there must be assignments to the outputs between the declaration and the call. The return variable is a single location that is present following a `call` instruction or whenever an assignment is made to it; its type is determined by a flow analysis. Each input, standard place, local, output, and return is a LIL variable, and is referred to using the names `i0`, `i1`, ..., `sp0`, `sp1`, ..., `l0`, `l1`, ..., `o0`, `o1`, ..., and `r`, respectively.

All LIL variables and operations are typed by a simple type system. The type system makes just enough distinctions to know the width of values and where they should be placed in a given calling convention. For example, the type system distinguishes between floating-point and general-purpose values but not between signed and unsigned. In addition, the type system distinguishes various kinds of pointers (*e.g.*, pointers to heap objects versus pointers to fixed VM data structures), because in the future we may want the LIL code generator to be able to enumerate garbage-collection roots on LIL activation frames. The complete list of types is: `g1`, `g2`, `g4`, and `g8` for general-purpose values; `f4` and `f8` for floating-point values; `ref` and `pint` for pointer values; and `void` for return types only (nothing returned).

The LIL language includes a validity checker. This check makes sure that each stub is sensible, and can catch some basic errors. The conditions checked include the following. All labels used must be declared exactly once. The last instruction cannot fall through, it must jump or return (both tail calls and no return calls satisfy this requirement). Every control flow path to a point must set up the activation frame in consistent ways (*i.e.*, same number of locals, number and type of outgoing arguments, presence or absence of an `M2nFrame`, *et cetera*). This last condition is checked by a straightforward dataflow analysis. The validity checker also imposes certain restrictions on the program that make code generation easier. Space prevents a detailed description of the validity check.

Syntactically, a LIL stub consists of an entry declaration followed by a sequence of other declarations and instructions. The declarations and instructions are described in the following subsections, except that we first describe what standard places are. This section ends with a brief description of the implementation of LIL in ORP.

3.2.1 Standard Places

Standard places are a set of implicit arguments, which can be passed to a stub in addition to the normal arguments passed via the regular calling conventions. To see why standard places are necessary, consider the example of *compile-me* stubs. When a new class is loaded, its methods do not need to be compiled immediately. Instead, a compile-me stub is installed. When the method is invoked for the first time, the compile-me stub causes the method to be compiled by the JIT, after which the compiled code is invoked with the original arguments.

Conceptually, the compile-me stub for an instance method taking an integer and returning a float does the following: It pushes an `M2nFrame` in case garbage collection occurs or exceptions are thrown during compilation. Then it calls the VM to compile the method, passing a pointer to the VM data structure representing the method. This function returns a pointer to the native code, the stub pops the `M2nFrame`, and then performs a tail call to the native code. Here is the LIL code that achieves this:

```
entry 0:managed:ref,g4:f4;
push_m2n;
out platform:pint:pint;
o0=method;
call jit_method;
pop_m2n;
tailcall r;
```

where `method` is a pointer to a VM data structure representing the method in question.[5] Except for the `entry` declaration and the value of `method`, the rest of the code above is common for all compile-me stubs. Therefore it would be desirable to factor this code out into a separate, generic stub. Compile-me stubs could then call this stub and pass it the value of `method` as an argument. However, passing an argument in the usual way would upset the argument stack, which is already holding the arguments of the method to be compiled.

The easiest and most efficient way to pass this argument between the two stubs is to use a separate argument-passing mechanism that does not interfere with the call stack. Standard places serve exactly this purpose. Passing information between stubs through standard places in LIL is roughly analogous to passing information between functions through

[5]ORP's compile-me stubs also contain some exception re-throwing code not shown here.

Table 1: LIL Declarations

LIL syntax	Description
`entry n:cc:Ts:RT;`	Stub signature
`entry n:cc:arbitrary;`	Stub signature
`out cc:Ts:RT;`	Call setup
`in2out cc:RT;`	Call setup
`locals n;`	n locals
`std_places n;`	n standard places
`:label;`	Label declaration

global variables in C. In the example, the method-specific stub becomes:

```
entry 0:managed:ref,g4:f4;
std_places 1;
sp0=method;
tailcall compile_me_generic;
```

The declaration `std_places 1;` creates a standard place. The next instruction assigns `method` to it. The tail call passes this extra argument to `compile_me_generic` using fixed scratch registers. The generic compile-me stub is the following:

```
entry 1:managed:arbitrary;
push_m2n;
out platform:pint:pint;
o0=sp0;
call jit_method;
pop_m2n;
tailcall r;
```

The entry declaration declares that one extra argument will be passed in a standard place. This standard place is then used in the third instruction `o0=sp0;` as the argument to the call to `jit_method`. Notice also that the entry declaration contains the keyword `arbitrary` instead of parameter and return types. This means that the stub should work for any number and type of parameters and type of return. Such a stub cannot access the inputs and cannot return—it must end with an unconditional jump, a call that does not return, or a tail call.

The need for this extra argument-passing mechanism is important to several classes of stubs. The standard-places mechanism is one aspect of LIL that is different from traditional compiler intermediate representations.

3.2.2 Declarations

LIL declarations are summarized in Table 1. An `entry` declaration must appear at the beginning of every stub, and only there. The other declarations can appear anywhere after the entry declaration, and take effect at the point where they appear. An entry declaration contains a colon-separated list of items. The first item is the number of standard places passed to the stub. The rest is the stub's signature. A *signature* comprises a calling convention, a comma-separated list of parameter types, and a return type. As well as the `managed` and `platform` calling conventions shown in the examples, there is `jni` for the calling convention specified by JNI [10], `rth` for runtime helpers, and `stdcall` (for the `stdcall` calling convention, needed for CLI). In some stubs, such as the generic compile-me stub, the keyword `arbitrary` is used instead of parameter and return types in the entry declaration. This means that the stub works for an arbitrary number of parameters of arbitrary types and an arbitrary return type. As previously mentioned, such a stub may not access input variables nor return.

The `out` and `in2out` declarations set up an output area for a subsequent call. The `out` declaration is followed by a signature for the function being called. The `in2out` declaration is followed by the calling convention and return type; the number and types of arguments to the callee are the same as for the stub. Strictly speaking, `in2out` is both a declaration and an instruction. In addition to setting up for the call, it also copies the input variables to the output variables. This may be more complicated than it seems, since the stub and the callee may follow different calling conventions. On the IA-32 architecture, for example, an `in2out` instruction inside a `managed` stub calling a `platform` function will need to reverse the order of arguments on the stack.

The instruction `locals n;` declares `n` local variables, which are then available in subsequent instructions. Similarly, `std_places n;` creates `n` standard places. Finally, `:l;` declares a label `l`.

3.2.3 General-Purpose Instructions

LIL includes instructions typically found in a low-level compiler intermediate representation for doing arithmetic, loading, storing, and control flow. These appear in Table 2. Instructions that are specific to a VM implementation, and ORP in particular, are described in the next section.

The arithmetic instructions include unary and binary operations common on CPUs, such as addition, subtraction, negation, bitwise operations, sign and zero extension, shifts, *et cetera*. In Table 2, `v` stands for a variable (*i.e.*, input, output, local, or standard

Table 2: LIL General-Purpose Instructions

Category	LIL syntax	Description
Arithmetic	`v = o;`	Move
	`v = uop o;`	Unary
	`v = o1 op o2;`	Binary
Memory	`ld v, addr;`	Load
	`st addr, o;`	Store
	`inc addr;`	Increment
	`cas addr=o1,o2, label;`	Atomic cmp and swap
Calls	`call o;`	Ordinary
	`tailcall o;`	Tail call
	`call.noret o;`	No return
	`ret;`	Return
Branches	`jc cond, label;`	Conditional
	`j label;`	Always

Table 3: LIL VM-Specific Instructions

Category	LIL syntax
M2nFrames	`push_m2n o(,handles)`$^?$`;`
	`pop_m2n;`
	`m2n_save_all;`
JNI Handles	`handles = o;`
Thread Pointer	`v = ts;`
Allocation	`alloc v,n;`

place), and `o` stands for an operand (*i.e.*, a variable or immediate value). In addition, a coercion can be applied to an operand to change its type to an equivalent one. On a 32-bit architecture, `g4`, `ref`, and `pint` are equivalent; on a 64-bit architecture, `g8`, `ref`, and `pint` are equivalent. A coercion has the form `:T` following the variable or immediate.

The general form for loads and stores is `ld v,addr;` and `st addr,o;`. There is also a memory increment operation, `inc addr;`. This operation is used by stubs to increment statistics and performance counters. An address consists of an optional base variable, an optional index variable, a byte offset (which can be zero), and a type. The type is the type of the memory location being accessed. The index variable can have a scale of one, two, four, or eight. Both the complex address form and the `inc` instruction match well with the IA-32 architecture's memory operations and can easily be expanded to an efficient sequence of IPF instructions. In addition, an address can specify acquire or release semantics and load instructions can specify zero or sign extension of subword values.

There is also an atomic compare-and-swap operation, `cas addr=o1,o2,label;`, for use in synchronization stubs. It compares the first operand against the memory value given by the address. If these values are equal it stores the second operand into the same memory address; otherwise it jumps to the label. The whole operation appears atomic to all threads.

The examples showed uses of call instructions, `call` and `tailcall`. The form `call.noret` is like `call` except that the LIL writer is asserting that the function does not return. The LIL code generator will not generate native code beyond the actual call instruction. It also affects the validity of a LIL stub—`call` cannot end a stub but `call.noret` can. This form is commonly used for exception throwing functions.

We have already seen examples of conditional jumps. The conditions are formed using the common relational operators (*e.g.*, `==` and `<=`). LIL also has unconditional jumps.

3.2.4 ORP-Specific Instructions

The `push_m2n` instruction sets up an `M2nFrame` on the stack. It includes an operand and an optional `handles` keyword. The operand is a pointer to a VM data structure for a method. If the stub interfaces to a native method, the operand should be this native method. All other stubs should use NULL. The `handles` keyword is explained below. The instruction `m2n_save_all;` saves additional information to the `M2nFrame` needed for exception propagation. This information is not saved automatically by `push_m2n`, because doing so is expensive on IPF, and it is very often unnecessary.

All `M2nFrames` also include a list of JNI handles. These handles are a GC-safe mechanism for referencing objects. The `handles` keyword in the `push_m2n` instruction declares that such handles will be used, and the matching `pop_m2n` instruction will free them. Otherwise the freeing step is skipped. The instruction `handles=o;` sets the list pointer in the `M2nFrame` to the operand. This instruction just sets the pointer to a list of handles in the frame; other stub code is responsible for creating the necessary handle structures.

The allocation instruction `v=alloc n;` allocates `n` bytes of memory on the stack and places a pointer to it into `v`. This space is available until the stub returns. Stubs use this instruction to create structures, such as JNI handles, and to pass pointers to them.

1.	entry 0:managed:f8,f8:f8;
	push_m2n method,handles;
	locals 2;
	alloc l0,8;
2.	ld l1,[mi:ref];
	st [l0+0:ref],l1;
	st [l0+4:pint],0;
	handles = l0;
3.	out platform::void;
	call gc_enable;
4.	out jni:pint,pint,f8,f8:f8;
	o0 = jni_env;
	o1 = l0;
	o2 = i0;
	o3 = i1;
	call Java_java_lang_Math__pow;
	l1 = r;
5.	out platform::void;
	call gc_disable;
6.	l0 = ts;
	ld l0,[l0+ceo_offset:ref];
	jc l0=0,no_exception;
7.	m2n_save_all;
	out platform::void;
	call.noret rethrow_current_exception;
8.	:no_exception;
	r=l1;
	pop_m2n;
	ret

Figure 3: LIL code sequence for the JNI wrapper to `java.lang.Math.pow`.

3.2.5 JNI Stubs

To illustrate some of the allocation and `M2nFrame` setup features, this section discusses JNI stubs, which are the most complex stubs in ORP. These stubs are responsible for a transition from managed code to native methods implemented according to the JNI specification [10]. They need to perform a number of operations, including matching calling conventions, enabling garbage collections, handling synchronization, rethrowing exceptions, and converting references to and from handles.

As an example, consider `Math.pow`, which is a static native method that takes two doubles and returns a double. Figure 3 shows the LIL code that interfaces managed code to a JNI version of `Math.pow`. In the figure, `method` stands for the VM data structure for `Math.pow`, `mi` stands for the address of the field in the VM data structure for `Math` that points to the `java.lang.Class` object for class `Math`, `jni_env` stands for the JNI environment structure in the VM, and `ceo_offset` stands for the offset of the current exception object in the thread structure.

The first block declares the inputs and locals, pushes an `M2nFrame`, and allocates 8 bytes of space for storing an object handle on the stack. The second block adds the `Math` class object to the list of local object handles. The third block calls the VM's `gc_enable` function to allow garbage collection to occur while the thread is subsequently executing within native code. The fourth block calls the actual native method. The fifth block calls `gc_disable` prior to the return to managed code. The sixth block tests whether the native method threw an exception, branching to the `no_exception` label if not. Otherwise, the seventh block calls the VM function `rethrow_current_exception` to propagate the exception. The final block pops the previously pushed `M2nFrame` and returns to managed code.

In general, generating JNI stubs is a template-driven process. Some snippets of code are included in a JNI stub only for certain types of methods. For example, a JNI stub will include a synchronization snippet only if the corresponding method is declared `synchronized`. In fact, there are two different synchronization snippets, for static and non-static methods. Other code pieces are repeated for each of a method's arguments. Actually, different pieces of code are used for reference and basic-type arguments. JNI stub generation cannot be expressed as a simple "fill in the blanks" process. It actually takes hundreds of lines of C code to generate the LIL source code for each JNI stub.

3.3 Implementation

The LIL system has two parts: a parser and a code generator. The parser takes a C string as input and produces an intermediate representation (IR) of the LIL instructions. The code generator takes the LIL IR as input and produces machine instructions for a particular architecture.

The parser includes a `printf`-like mechanism for injecting runtime constants such as addresses of functions or fixed VM data structures. For example, here is the C code that generates the method-specific compile-me stubs we encountered in Section 3.2.1.

```
NativeCodePtr
  create_compile_me(Method_Handle m)
{
  LilCodeStub* cs = lil_parse_code_stub(
    "entry 0:managed:%0m;"
    "std_places 1;"
    "sp0=%1i;"
    "tailcall %2i;",
    m,
    m,
    create_compile_me_generic());
  assert(lil_is_valid(cs));
  NativeCodePtr addr =
    LilCodeGenerator::get_platform()->
      compile(cs);
  lil_free_code_stub(cs);
  return addr;
}
```

Our code generators make a couple of prepasses to gather information and decide where to locate LIL variables, and then a main pass to generate the code. All these passes are simple sequential scans of the instructions. Each LIL instruction is translated into a sequence of machine instructions based on information gathered in the prepass. No sophisticated optimizations, intermediate languages, or peephole optimizations are used, although we are considering whether a smarter instruction scheduler and template packer would improve performance on the IPF architecture.

LIL is very lightweight: The parser and infrastructure for LIL is about 3500 lines of code, the IA-32 code generator is about 1500 lines of code, and the IPF code generator is about 2200 lines of code. The code generators also use the `M2nFrame` modules, which are about 300 lines for each architecture, and the code emitters, which are 2200 and 13500 lines respectively. However, the `M2nFrame` modules and the code emitters would be needed even without LIL.

4 Benefits

The motivation for creating LIL was to improve the portability, maintainability, and correctness of stubs. These benefits are described in Sections 4.1 to 4.4. Perhaps surprisingly, using LIL can also improve performance. Section 4.5 describes how LIL can be inlined, or *implanted*, into JIT-compiled code to make ORP's runtime support system more efficient.

4.1 CPU Independence

The most direct benefit of implementing stubs using LIL is that LIL is architecture-neutral. Each stub is written only once, instead of being reimplemented for every platform. In addition, any performance enhancements or bug fixes applied to a stub are automatically propagated to all architectures. Such modifications have been common during ORP's development. Before we switched to LIL, machine-language stubs on the IA-32 and IPF architectures diverged over time, sometimes to the extent of implementing slightly different functionality. LIL stubs do not suffer from such problems.

4.2 OS Independence

Several high-performance stubs operate directly on data structures that depend on the VM and OS. Of these data structures, the two most important are thread-local storage and `M2nFrames`. LIL hides the implementation of these constructs, and allows stubs to access them in a platform-independent manner.

Loading the thread pointer is needed for efficient implementation of object allocation and synchronization, which are significant to the overall performance of ORP. As mentioned in Section 2.3, this operation is requires different sequences on Windows versus Linux, and IA-32 versus IPF platforms. As we saw in Section 3, LIL includes a primitive for loading the thread pointer. This enables us to use the same stubs on all platforms and operating systems. Any operating system dependences are hidden within the LIL code generator.

4.3 Readability

Because LIL is more high-level than machine language, most stubs become more concise and readable when expressed in LIL. This is particularly true for complex stubs, such as the ones used to implement JNI, described in Section 3.2.5. ORP's IA-32 implementation of JNI stubs has about 700 lines of C code; ORP's IPF implementation has about 400 lines of C code. The equivalent LIL implementation is about 320 lines of C code, and is also much easier to understand and modify. In particular, the IA-32 non-LIL implementation of JNI contains extensive debugging code, because certain aspects of setting up JNI stubs were especially error-prone. One such aspect is accessing the call stack; since the stub needs to push things onto the stack, offsets of stack values keep changing during the stub's execution. Also, transitioning between managed and JNI code on the IA-32 architecture requires reversing the order of arguments on the stack, which is particularly tedious. LIL code, on the other hand, sets up the stack with simple statements such as `o3=i2;` and offsets and argument orders are automatically taken care of by the LIL code generator.

4.4 Correctness

Implementing stubs directly in machine language makes it hard to ensure their correctness. In addition to being difficult to read, machine language offers no mechanisms for automated validity checking. This is especially problematic, since even small bugs in stubs are almost certain to break the system.

Using LIL helps ensure correctness in two ways. First, some of the most tedious conventions are abstracted away by LIL. For example, the stub implementor no longer has to worry about implementing calling conventions, about which system values are in what registers or memory addresses, or about which machine registers are available for storing local variables. The implementation of such conventions needs to be checked only once, in the LIL code generator.

Second, the LIL code generator can check LIL source code for consistency, as explained in Section 3.2. Although LIL is not a strongly typed language, the limited semantic checking it supports can still catch some of the most frequent bugs, such as providing the wrong number of arguments to a function, accessing incoming arguments that do not exist, or returning without setting the return variable. In a sense, the semantic checker of the LIL language is roughly equivalent in power to that of C. This is a big improvement over using assembly language, which has no semantic checking whatsoever.

4.5 Code implants

We previously discussed LIL's value as a tool for implementing stubs that are called by JIT-generated code. But these calls introduce some inefficiencies because many stubs, including those for runtime type identification and memory allocation, are called so frequently. One way to avoid this call overhead, while maintaining ORP's strict interface between the JIT and the VM, is to move toward an *implant-based* runtime support system. In such a system, the VM passes LIL sequences to the JIT instead of translating the sequences itself. The JIT then translates LIL sequences to its own intermediate representation, implanting them into the code it generates. Much like normal method inlining, code implanting not only avoids call overheads, but also exposes more opportunities for optimization.

In general, a LIL-based code implanting system would work as follows.

1. The JIT first makes a runtime support request to the VM. Along with the request, it passes context information about the stub's call site. Such information might include which stub arguments are constant or NULL, the profile weight of the call site, *et cetera*.

2. In response to that request, the VM generates a pre-optimized LIL stub using the context information and its knowledge of its own data structures and other implementation details.

3. The JIT receives back from the VM a snippet of LIL code instead of a stub address. The JIT can then translate the LIL code to its own intermediate representation. This translation is straightforward for LIL's general-purpose instructions, but not possible for ORP-specific instructions (see Section 3.2.4). Instead, the JIT can treat such instructions as black boxes.

4. During the code generation phase, the JIT can ask the VM to expand each black-box instruction to machine code.

A preliminary version of a LIL-based code implanting system has been implemented in ORP. This system implants the VM stubs for the `checkcast` and `instanceof` bytecodes into the *O3 JIT*, which is currently ORP's best-performing IA-32 JIT. Experiments using this system show that implanting just these two stubs improves overall performance by 3% for the SPEC JVM98 [14] suite. The current system implements only part of the scheme described above. A full-fledged system would be able to implant more stubs and would have more optimization opportunities. We expect the final implanting system to provide significantly greater performance improvements.

LIL code implants, their implementation, and their performance benefits are described in more detail by Cierniak *et al.* [6].

5 LIL Performance

In designing LIL, our goal was to obtain the benefits described in the previous section without sacrificing ORP's excellent performance. Since ORP previously implemented all its stubs through hand-crafted machine-code generation, it is possible to compare the performance of LIL stubs to that of hand-coded stubs. This section will present this comparison and show that we have mostly retained performance. Figure 4 lists the LIL stubs that have been written so far, and the stubs for which LIL versions have not yet been written. Of the non-LIL stubs, the CLI stubs and atomic compare-and-swap stubs should be straightforward—we have not gotten around to them yet. The other two stubs require additional features for LIL that are highly specific to those two stubs. Having the hand-coded assembly versions of these stubs is roughly equivalent to the additions that would be needed to the LIL code generator to support them.

We evaluate the overhead of LIL by measuring ORP with hand-coded assembly versions of all stubs versus ORP with LIL versions of the stubs, on both

LIL Stubs
- **Compilation**
  - Compile-me generic, compile-me specific, recompile
- **Native-method interface**
  - JNI and PInvoke stubs
- **Exceptions**
  - Throw, lazy throw, throw specific exceptions, throw linking exception
- **Allocation**
  - New object, new array, multinewarray, load constant string
- **Synchronisation**
  - Monitor enter and exit for objects and classes
- **Type tests**
  - Checkcast, instanceof, aastore
- **Arithmetic helpers**
  - Float to integer, double to integer, long shifts, long multiplies, long divides, *et cetera*
- **Miscellaneous**
  - Load interface vtable, initialise class, character-array copy

Non-LIL Stubs
- Native to managed transition
- Transfer control to exception handler
- CLI-delegate stubs
- CLI unboxers
- Atomic compare and swap

Figure 4: LIL and non-LIL Stubs

the IA-32 and IPF architectures. We make a few exceptions for several performance-critical stubs, whose assembly versions have been highly tuned for their particular platforms. These are the object monitor enter and exit stubs on the IA-32 architecture, and the new object, new array, and character-array copy stubs on the IPF architecture.

The IA-32 performance numbers were obtained on a 4-processor, 2.0 GHz Intel® Xeon^{TM} processor-based machine with HyperThreading disabled, with 4 GB physical memory, and running Windows 2000 Advanced Server. The IPF performance numbers were obtained on a 4-processor, 1.5 GHz Itanium® 2-based system with 6 MB L3 cache and 16 GB physical memory, running Windows Server 2003, 64-bit edition.

We measured the performance of the seven components of SPEC JVM98 [14] as well as SPEC JBB2000 [15].[6] We used a 96 MB heap for the SPEC JVM98 components on both architectures, 1 GB for SPEC JBB2000 on the IA-32 architecture, and 4 GB for SPEC JBB2000 on the IPF architecture. In general, SPEC JVM98 models client-side applications, which do not typically require significant amounts of memory. SPEC JBB2000 is designed to model more memory-intensive server applications, hence the larger heap sizes.

Figure 5 shows the relative speedup of ORP with LIL stubs over ORP with only assembly stubs on both the IA-32 and IPF architectures. On the IA-32 architecture, LIL either has no effect or improves performance for half of our benchmarks, allowing for a 0.5% experimental noise margin, and shows no more than a 7% degradation for the rest. LIL is even more promising on the IPF architecture, with less than 4% slowdown for jess and db.

When all available LIL stubs are used, the slowdowns increase substantially, particularly for SPEC JBB2000. This is because the few assembly stubs included in Figure 5 have been highly tuned for their particular platforms. The IA-32 tuning includes the use of instructions that are more efficiently implemented on the Intel® Xeon^{TM} processor. The IPF tuning include instruction scheduling and the use of some specialized instructions. We are currently investigating whether comparable tuning is possible in LIL. The work described here has allowed us to identify the critical stubs and thereby focus the LIL optimization efforts on a few architecture-specific issues.

6 Related Work

The Jikes RVM [3] is implemented almost entirely in Java. In order to support the unsafe low-level operations it needs to directly access memory, machine registers, and operating-system resources, Jikes includes "escape" mechanisms to circumvent Java's type system and memory model. Its primary escape mechanism is the `Magic` class. This includes static methods to do typecast, compare-and-swap, and memory fence operations, read and write memory, transfer control to specified addresses, and access stack frame contents (*e.g.*, caller's frame pointer, next instruction address). Each JIT for the Jikes RVM treats a call on a method of the Magic class specially: it implements the call using a sequence of inlined machine instructions. Although support for Magic and the other Jikes escape facilities must

[6]We use the SPEC benchmarks only as benchmarks to compare the performance of the various techniques within our own VM. We are not using them to compare our VM to any other VM and are not publishing SPEC metrics of any kind.

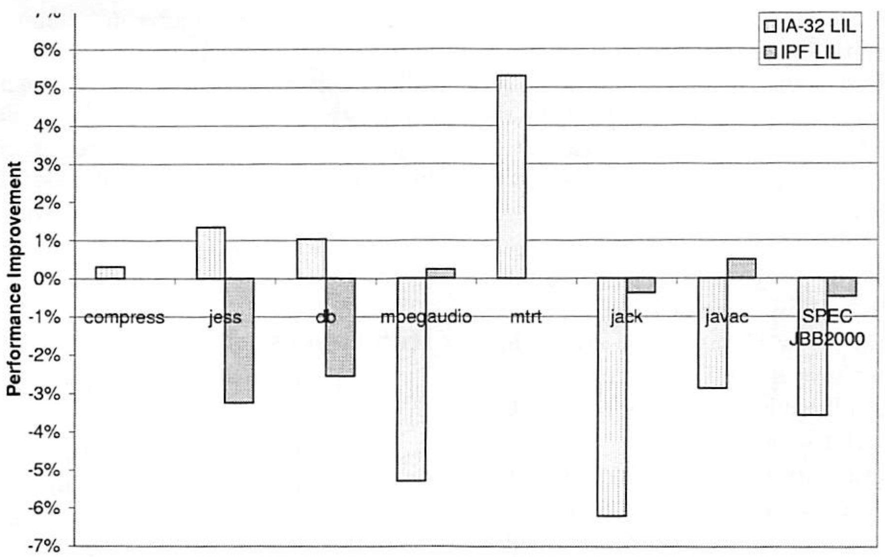


Figure 5: Performance impact of LIL.

be added to each new compiler, their implementation is relatively simple. The Magic class can be used to implement much of the same functionality as LIL. However, Magic is lower level: its operations are close to machine instructions. This makes code that uses Magic less readable. It also makes it easy to make mistakes, which can corrupt the internal state of the VM. Static analyses [12] have been implemented to check code using Jikes escape mechanisms for GC-pointer safety (that the system can determine at every GC point all the locations that contain references), but these analyses do not check for as many errors as the LIL code generator does. In addition, code using the Jikes escape mechanisms must often be reimplemented for each new machine architecture.

Griesemer [9] describes the facility used by the Hotspot virtual machine to generate machine code at runtime. C++ *Assembler* classes are used by the VM to emit stubs and other machine code: the VM calls an Assembler method to generate each machine instruction, to define labels, and other operations. So, a sequence of these calls are a kind of specification for a machine code sequence. Like LIL, the instruction-emitting Assembler methods are slightly higher-level than raw machine instructions since calling them may actually generate different instructions depending on which member of a processor family the application is running on. They also provide full control over the emitted code: assuming that the appropriate instruction-generation methods have been defined, any machine code sequence can be defined. However, these Assembler methods are architecture-specific, so different sequences of calls are needed for the IA-32 architecture versus for the IPF architecture. They are also extremely low-level, which makes it hard to understand or maintain the stub-generation code. ORP has a similar set of instruction-emitting classes for each processor architecture, and these were used before we developed LIL.

Microsoft's Shared Source Common Language Infrastructure [13] includes an interpreted language, ML, that is used to implement marshaling stubs executed during RPC calls or when transitioning between native library code and either the VM or JIT-generated code. ML includes opcodes to load and store values into marshaling buffers, convert values, and throw exceptions on errors. Unlike LIL, however, this language is specialized: it supports only argument marshaling.

Remote procedure calls (RPCs) simplify the construction of distributed programs. Part of this ease is the transparency of RPCs to the programmer, and this transparency requires the automatic construction of stubs to make calls across machines look like ordinary calls. Such stubs are similar in some ways to the native-method invocation stubs, and some of the issues are the same. However, they are different enough that the same solutions are not applicable.

7 Future Work

Many of LIL's features are the result of our experience using LIL to rewrite stubs that were formerly hand-coded. Such features include LIL's support for loading the thread pointer, and its support for the `M2nFrames` that mark the transition between managed and native frames on the stack. However, LIL is not currently able to implement all the stubs required by ORP. We are considering whether it is worth while adding a couple of new features to LIL to support the two stubs that we cannot write in LIL.

Overall, we find the performance of LIL on both the IA-32 and IPF architectures satisfactory. However, there are still a few highly tuned stubs that LIL significantly under performs. Future work will try to improve the LIL code generators, so that LIL achieves similar performance.

Our early experiments on implanting LIL stubs into JIT-generated code have been very encouraging. We intend to continue these experiments and further develop the code implanting system. Other VM-emitted code sequences will be converted to LIL and then passed to the JIT for inlining and optimization. This will require the development of a general way for the JIT to inline LIL sequences. It will also be necessary to add to LIL any features required for implants.

8 Conclusions

The ORP managed runtime environment executes Java and CLI applications. It manages to combine high performance with modularity and experimentation ease. This is in part because of its use of optimized runtime support stubs, which are dynamically created by the VM to implement such common operations as object allocation, exception throwing, and native-method invocation. Problems with creating stubs using hand-written machine code led us to develop a new language, LIL, for specifying them. LIL is architecture-neutral and high-level than machine code, yet provides enough low-level facilities to achieve good performance and to allow low-level operations such as call-stack manipulations and register access.

We found LIL stubs to be shorter and more readable than the processor-specific machine-code sequences they replace. This has simplified their maintenance and has made it easier for us to experiment with new optimizations. We studied the performance impact of LIL stubs using SPEC JVM98 and SPEC JBB2000 running on ORP on both the IA-32 and IPF platforms. Our results demonstrate that using LIL retains most of the performance of hand-written stubs. That is, using LIL offers significant software-engineering advantages without significant performance losses.

While LIL is still under development, our experience suggests that both its performance and capabilities will continue to improve. We are particularly optimistic about the potential of *code implants*, LIL sequences passed to the JIT for optimization and inlining into the JIT-generated code. These implants not only avoid the overhead of stub calls, but expose further opportunities for optimization such as instruction reordering and scheduling.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art02_starjit/p01_abstract.htm`.

[2] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1), February 2000.

[4] A. Bik, M. Girkar, and M. Haghighat. Experiences with JAVA JIT Optimization. *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, October 1998.

[5] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm`.

[6] M. Cierniak, N. Glew, S. Triantafyllis, M. Eng, B. Lewis, and J. Stichnoth. Object-

Model Independence via Code Implants. *Proceedings of the Workshop on Multiparadigm Programming with OO Languages (MPOOL'03)*, October 2003.

[7] M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[8] ECMA. *Common Language Infrastructure.* ECMA, 2002. Available at `http://www.ecma-international.org/publications/Standards/ecma-335.htm`.

[9] R. Griesemer. Generation of virtual machine code at startup. In *Proceedings of the OOPSLA '99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design.* Sun Microsystems, Inc., November 1999.

[10] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[12] J.-W. Maessen, V. Sarkar, and D. Grove. Program analysis for safety guarantees in a java virtual machine written in java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 62–65. ACM Press, 2001.

[13] Microsoft. Shared source common language infrastructure. Published as a Web page, 2002. See `http://msdn.microsoft.com/net/sscli`.

[14] Standard Performance Evaluation Corporation. SPEC JVM98, 1998. See `http://www.spec.org/jvm98`.

[15] Standard Performance Evaluation Corporation. SPEC JBB2000, 2000. See `http://www.spec.org/jbb2000`.

Detecting Data Races using Dynamic Escape Analysis based on Read Barrier

Hiroyasu Nishiyama
Systems Development Laboratory, HITACHI, Ltd.
1099, Ohzenji, Asao-ku, Kawasaki-shi, 215-0013 Japan
nisiyama@sdl.hitachi.co.jp

Abstract

In multi-threaded programs, a data race results in extremely hard to locate bugs because of its non-deterministic behavior. This paper describes a novel dynamic data race detection method for object-oriented programming languages. The proposed method is based on the lockset algorithm. It uses read-barrier-based dynamic escape analysis for reducing number of memory locations that must be checked at runtime for detecting data races.

We implemented the proposed data race detection method in HotSpot Java[1] VM. The results of an experimental evaluation show a significant performance improvement over the previous write-barrier-based method and also that the proposed method can perform data race detection with a relatively small runtime overhead.

1 Introduction

Compared with traditional sequential applications, multi-threaded applications are prone to errors arising from the concurrent nature of program execution. A *data race*[22] is one of the primary causes of unpredictable behavior in multi-threaded programs. Data races occur if two parallel threads access a same memory location without using implicit or explicit synchronization that prevents simultaneous access of the data; and if at least one thread is write access. A program involving data races is sensitive to the dynamic state of its runtime environments such as in thread scheduling or user interactions. This leads to unpredictable behavior of the program. Unfortunately, the cause of this problem is difficult to pinpoint.

[1]Java and other names are trademarks or registered trademarks of SUN Microsystems Inc. in the United States and other countries.

Java[16, 28] is a widely used object-oriented programming language that includes support for multi-threading. Java provides thread functionality as a means for describing concurrently executing entities. Threads in Java are actively utilized by applications such as GUI applications and Internet servers.

Java provides language-level synchronization constructs based on the *monitor* as a means for serializing accesses from concurrently executing threads. Language-level integration of the monitor in Java simplifies the specification of critical sections. Although this may decrease the occurrence of synchronization programming errors, the complete prevention of synchronization errors is still hard to achieve for large-scale multi-threaded applications.

Extensive studies on optimizations to eliminate useless synchronizations[15] and on reduction in overhead for uncontended cases[4, 5] have been made; however, synchronization specifications are usually considered to be a source of severe performance overhead. Consequently, programmers sometimes are conservative about using synchronization. This may introduce additional unintended synchronization errors.

Many works have treated automatic detection of data races; and there exists several commercial or open-source products that detects data races[13, 17]. From a theoretical perspective, Netzer and Miller proposed a classification of data races[22]: *actual race*, *feasible race*, and *apparent race*. They have also shown that a general data race detection is NP-hard. Past research on automatic data race detection can be classified as ones involving static, dynamic, or combined methods. The static approaches include an approach for adding annotations to a program and proving the correctness with a theorem prover[18], a post-mortem approach that analyzes events log of a program execution[3], and an approach using a language extension incorporating a type system for static race detection[10]. Dynamic approaches examine the access pattern to shared locations on-the-fly verifying its correctness according to some criteria such as the *locking discipline*[6] or *happens-before* relation[19].

In this paper, we propose a novel on-the-fly data race detection method for object-oriented languages based on the *lockset algorithm*. The lockset algorithm dynamically verifies the locking discipline to detect synchronization errors in a given program.

Although the lockset algorithm has been applied to real-life applications, it incurs a severe performance overhead. Our method reduces this overhead by exploiting the intuitive characteristic of program behavior that most objects in a typical object-oriented application do not incur shared access from multiple threads.

We use this property for reducing the number of objects to be examined while a program is being executed. The number of examined objects is limited by using *dynamic escape analysis* based on the *read barrier*. Dynamic object reachability tracking using the read barrier enables a reduction in the number of shared objects compared with the previously proposed write barrier based method.

The rest of the paper is organized as follows. Section 2 summarizes related works on data race detection based on the lockset algorithm. Section 3 describes the new data race detection approach using the read barrier. Section 4 describes its implementation in HotSpot Java VM. Section 5 describes our experimental results on a set of benchmark programs.

2 Detection of Data Races

Modern concurrent or multi-threaded programming frameworks provide structured synchronization primitives such as monitor or communication channel instead of much simpler ones such as spin lock or semaphore.

A monitor employed by Java is a programming language construct that encapsulates shared resources and its access functions. The monitor of Java can represent an exclusive program region using an acquisition and release of an object specified by the synchronized attribute of a method or a synchronized statement.

The lockset algorithm detects synchronization errors in a monitor-based concurrent program. This algorithm assumes that a correct program keeps the following locking discipline:

> *When some thread accesses shared data, the thread must hold at least one monitor. Furthermore, the intersection of the sets of all monitors held when accessing the data must not be empty.*

Let $L(v)$ represent a lockset for a shared memory location v, $E(v)$ represent a set of times when read or write reference events for the memory location v occurred, $T(v, i)$ represent a set of threads that accessed the memory location v at time i, and M_{ti} represent a set of locks that is held by thread t at time i. Then, $L(v)$ can be defined as follows:

$$L(v) = \forall_{i \in E(v)} \forall_{t \in T(v,i)} \cap M_{ti}$$

If the set $L(v)$ becomes empty, we can recognize that at least two accesses from different threads have been performed with nonuniform synchronization. Thus, we can suspect the occurrence of data races at the memory location v.

As an example of a program involving a data race, we will consider the program shown in Figure 1. The `Account` class in this figure defines the method `inc` which increments the field `balance` defined at (a), and also defines the method `dec` which decrements the field `balance`. Consider a case where thread #1 calls `inc` and thread #2 calls `dec`. The call for the `inc` method obtains a monitor with the synchronized statement at (b). Accordingly, the monitorset at the update point (c) of the `balance` is written as {`this`}. In regard to the call for `dec` by thread #2, since the method `dec` does not perform synchronization, the monitor set at update point (d) of the `balance` becomes empty. Calculating the lockset from the monitor set of each thread yields an empty set. Thus, we can suspect the possible occurrence of a data race for the field `balance` in this program.

```
                                                    Monitor Set
                                           ----------------------------------
                                           Thread#1   Thread#2   #1∩#2
class Account {
    private double balance;       // (a)    {this}       φ          φ
    ...
    void inc(double diff) {
        synchronized(this) {      // (b)
            balance += diff;      // (c)    {this}
        }
    }
    void dec(double diff) {
        balance -= diff;          // (d)                 φ
    }
}
-----------------------------
Thread#1: account.inc(...)
Thread#2: account.dec(...)
```

Figure 1: Example of a program containing a data race.

Eraser[26] is an example of a data race detection tool for pthreads based multi-threaded programs using the lockset algorithm. Eraser detects data races by using a binary rewriting technique to modify all memory reference instructions in a target program, and by monitoring memory references dynamically at runtime by calculating their associated monitor sets.

Since Eraser's target language, C or C++, can access any memory location through pointers, distinguishing data structures that need exclusive access from ones that only need thread private accesses is difficult. Thus, Eraser associates a data structure called *shadow memory* that exists in parallel with the program's accessible memory regions. In addition, it tracks all memory accesses dynamically in the shadow memory. This leads to a huge space and time overhead[2].

An object oriented programming language such as Java abstracts a main memory as a collection of objects instead of a simple sequence of bytes. Several studies have been done on using this characteristic to reduce the overhead of data race detection.

One example is the Object Race Detection[23] of Praun and Gross. It restricts units of data race detection at the level of objects instead of field elements. However, this restriction on race detection leaves open the possibility of false negative or false positive reports because of the ambiguity of field accesses or array element accesses. They also use static escape analysis[1] to decrease dynamic monitoring costs by excluding objects that can be proved as inaccessible from more than one thread.

The escape analysis is a technique for determining that an object is only accessible by a single thread. The escape analysis is also useful for optimizations such as synchronization removal[25], thread-local memory management[8], stack allocation of objects[11] and object inlining[7].

The data race detection system of Choi et al.[2] combines static escape analysis with compiler optimizations. For reducing detection costs, they restrict the detectable combination of data races. This means that their system does not report all access pairs that participate in data races, but guarantees that at least one access is reported.

The TRaDe system of Christiaens et al.[20] is a purely dynamic system based on happens-before relation. It calculates the set of objects that may be referenced from multiple threads by using dynamic escape analysis based on the write barrier.

In the related field of program analysis, Dwyer et al. described an combined approach of dynamic and static escape analysis of state-space reduction for model-checking object-oriented programs[9]. The model-checking is a software verification technique by exploring all possible execution sequences. Although the model-checking can provide more precise information of a program than the lockset algorithm, enumerating execution sequences is much heavy-weight operation.

[2]Savage et al. reported a 10X to 30X execution overhead[26].

3 Reduction in Data Race Detection Cost using Read Barrier

A data race detection method using the lockset algorithm monitors accesses to shared memory locations updating their monitor sets; it checks whether multiple threads accessed the location without obtaining common locks. However, monitoring all field references results in a severe performance penalty, as in Eraser. We reduce the penalty of reference monitoring using escape analysis by checking the reachability of an object from multiple threads. This reduction is accomplished by only tracking locksets for objects that are reachable from multiple threads.

We employed the dynamic method similar to TRaDe for our race detection system. The reason for using dynamic method is as follows: First, the dynamic method does not require complex whole program analysis and optimization framework. Second, the dynamic method can provide more accurate results than the pure static method because the static analysis can not follow exact data flow information. Finally, the dynamic class loading mechanism of Java makes an application of the static analysis difficult because it may invalidates assumption of the analysis. For program optimization, the deoptimization[12] can be used to cancel the incorrect compile time assumptions. In contrast, since a data race detection requires monitoring the history of a program execution, the invalidation of the assumption may result in false positive or false negative errors.

3.1 Dynamic Escape Analysis

An object in the Java virtual machine is only accessible from its allocating thread immediately after its allocation except for special objects such as class objects. Stack and method local variables are thread private resources and are not accessible from other threads[28].

Thus, an object O owned by a single thread becomes accessible from more than one thread when an O's reference or a reference to some object that can reach O by following reference chains is assigned to an object that can be referenced from multiple threads.

The data race detection method of TRaDe assigns each object one of the following two attributes: (a) a global attribute meaning that the object can be reached from multiple threads, and (b) a local attribute meaning that the object can be reached from only one thread. A globally reachable object at the point of its allocation such as class object is marked with a global attribute when it is allocated. When a reference R is assigned to an object of a global attribute, the global attribute is also assigned to the object that is pointed to by R and all objects that can be reached by following references from R.

The method of TRaDe can be regarded as a *write barrier*[24] based dynamic escape analysis method, which tracks object reachability at the point of its reference assignment.

We employed a read-barrier-based dynamic escape analysis method, which updates the reference attribute of each object when its reference is obtained from the object field.

This method reduces the data race detection overhead by calculating reference attributes of an object as follows:

1. A local reference attribute is assigned to an object after its allocation. A creating thread ID is also assigned to each object.
2. For each reference field in an object, if a reference attribute of its pointing object is local and its thread ID is not identical to the referencing thread, the reference attribute of the pointed object is changed to global.

The read barrier and write barrier are techniques that are usually used for maintaining intergeneration references in generational garbage collectors[24]. The write barrier is generally preferred for implementing a garbage collector because write references are less frequent than read references.

When using read or write barrier as a means for dynamic escape analysis, the costs of monitoring data races for global objects must be considered in addition to barrier overhead. Taking the following points into consideration, we can expect that the read barrier based methods will be superior to the write barrier based method:

- **Reduction in global attribute maintenance costs**

When a local object O_l is assigned to a reference field of a global object O_g, the write barrier based method needs to recursively traverse objects that can be reachable from O_l changing reference attributes of the traversed objects to global ones. This requires a high execution overhead on reference assignment and also requires extra memory overhead on the traversal.

The read barrier based method only needs to change the attribute of O_l to global when a thread which is not the owner thread of O_l has obtained a reference to O_l, in other words, when the object O_l has escaped to another thread. In contrast to the write barrier based method, the read barrier based method can perform data race detection with a much lower and definite cost. It also requires no recursive object traversals. Thus, we can expect lower runtime memory requirements.

- **Reduction in monitoring cost**

  The write barrier based method regards objects that can be reachable from global objects as potential candidates of data races. In contrast, the read barrier based method recognizes objects from which references are obtained by multiple threads, that is, objects reached by multiple threads through reference traversals, as potential candidates.

  Therefore, the target objects of the read barrier based method comprise a subset of the write barrier based method. As a result, we can expect a reduction in monitoring overhead of data race detection by using the read barrier.

3.2 Array Objects

A program such as a scientific application frequently uses a parallel processing idiom that divides arrays into multiple regions and assigns each region to separate worker threads. Since each worker thread of such an idiom usually accesses disjoint array regions, no data races can occur. However, since Java treats an array as a special kind of object, previous object based escape analysis consider the whole array as a target of the data race detection even for such an array-dividing idiom.

We extend the data race detection for an array to an array sub-block level to deal with such an idiom. An array is divided into fixed length sub-blocks and locality detection is applied to each array sub-block.

3.3 State Model

The basic idea of the read barrier based method is to classify objects and array sub-blocks into escaped and non-escaped groups at their reference field reading points. Our method follows Eraser[26] and Object Race Detection[23] and introduces an extended state model which mixes objects, array sub-blocks, and object fields as detection units.

Figure 2 shows the state model for an object, an array sub-block, and an object field. This model consists of the following seven states:

Object-specific states:

owned An object can be accessed from its allocating thread only. In the common case, when an object is created, it can be referenced from its creating thread only. Thus, the initial state of a normal object is owned.

escaped A reference to an object is obtained by threads other than its owner thread, that is, ones that have escaped to another thread. Fields of an escaped object should be monitored for race detection by the lockset algorithm.

The sub-block level states for an array are introduced after an array object changes its state from owned to escaped.

Field-specific states:

shared read A field in an escaped object is read by more than one thread. Since concurrent read requests for a field in a shared read state do not cause any data races, no conflict is reported.

shared modified A field in an escaped state is modified by some thread. No conflict is reported for a field with a non-empty lockset in the shared modified state because thread synchronization is expected to have been properly performed.

conflict A field of an escaped object is written by more than two threads or read and written by different threads with an empty lockset. A conflict is reported for the field, since a synchronization error for the field is suspected.

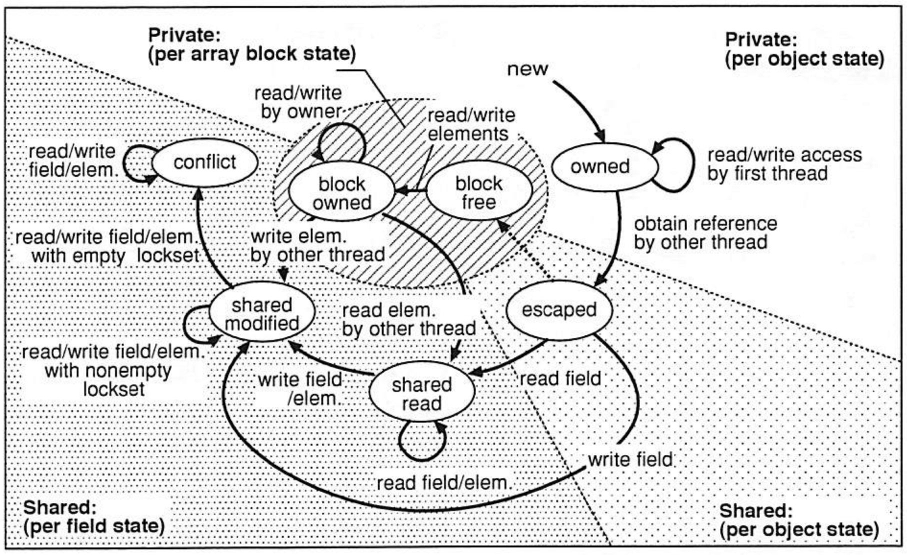


Figure 2: State Model for Objects, Array Blocks, and Fields

Array sub-block specific states:

block free This is the initial state of an array sub-block representing that an array object belonging to it has escaped to a non allocating thread. Since each sub-block element is free from accesses at the point of escape, the sub-block does not yet become subject to race detection by the lockset algorithm.

block owned An array sub-block in a block free state is referenced by some thread. The referencing thread becomes the owner of the sub-block. Following references for the array sub-block in the block-owned state by its owner thread does not initiate element-level data race detection. If a non owner thread accesses a sub-block in the block-owned state, the state of the block changes to the shared read or shared modified state and an element-level race detection by the lockset algorithm is initiated.

The read barrier based race detection method starts lockset computations for fields of an object when more than one thread can reach the object by traversing object references. Hence races may fail to be detected depending on dynamic thread scheduling conditions. However, as Savage et al. pointed out[26], many programs initialize objects without obtaining locks then pass their references to other threads. Therefore, starting a lockset computation after multiple threads reach an object can eliminate many false data race reports originating in object initialization.

4 Implementation

In this section, we describe an implementation of the proposed race detection method on the HotSpot Java virtual machine[21].

4.1 Object Format

Our implementation extends the object format of the base virtual machine, adding an extra field that is used for data race detection. Figure 3 shows the modified object format. The object format of the HotSpot Java VM consists of a two-word header: (a) a mark field that represents object age, hash code, and other information, and (b) a klass field that points to a class object. The modified object format is extended to a three-word header, adding a state field that represents reference attributes and external information about the object. The state field denotes two per object states (owned and escaped), two per array sub-block states (block free and block owned), and a special state for thread synchronization (locked). The owned and block-owned states are represented by making the LSB of the state field zero and recording thread ID to the remaining bit field. The difference between an owned and a block owned state is distinguished by object's class or the object kind implicitly recognized by the bytecode execution state. The escaped object state is represented by setting the LSB of the state field to one and the remaining state field to a pointer to the external per field information. The state field representation for an escaped array object is similar to the escaped object but it points to an array of sub-block information. Each element of the array keeps similar information on the state field for a normal object.

Three per field reference states (shared read, shared modified, and conflict) are recorded in an external data structure pointed to from the state field of an escaped object or the information array of an array sub-block.

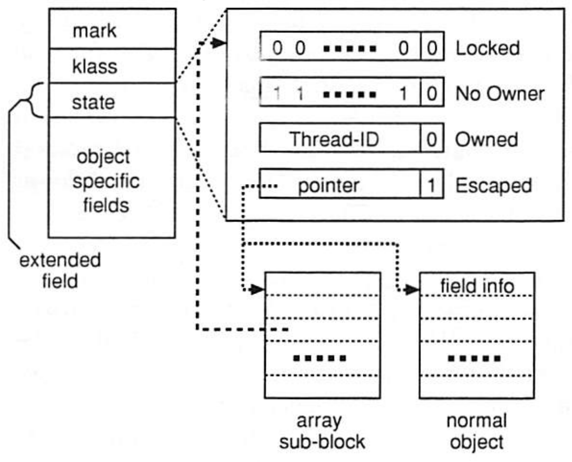


Figure 3: The extended object format including the state field which represents object reference state or points to external data structure.

4.2 Code Sequences

Bytecode execution of HotSpot VM is performed using indirect threading interpreter and dynamic JIT compiler. The threaded interpreter uses machine code sequences that is generated from code templates at interpreter startup.

To detect data races, we modified the machine code templates and compiler generated code sequences for the following bytecodes:

- `monitorenter/monitorexit`

  Machine code sequences for `monitorenter` and `monitorexit` bytecode are modified to record owned monitors in the per thread monitor stack. For the sake of efficiency, the conversion of a monitor stack into a monitor set is delayed until some object enters the escaped state.

- `getfield/getstatic/[ilfdabc]aload,` `putfield/putstatic/[ilfdabc]astore`

  The reading of the reference field by using `getfield` or `getstatic` bytecode is modified to check whether its target object has escaped or not.

  For each field reference or modification, if the target object has escaped attribute, the lockset algorithm is applied for detecting data races on the object. Result of the set calculation between locksets are cached for efficiency as is done by Eraser[26].

We also modified the field references by runtime system in the Java VM and the field references from native methods[3] through the JNI (Java Native Interface).

4.3 Dealing with Special Threads

A finalizer method may be defined in Java to allow work to be performed before an object is reclaimed by the garbage collector. In the HotSpot VM implementation, a finalizer is executed by a special thread called a finalizer thread[4]. The finalizer method is usually defined without explicit synchronization assuming implicit execution ordering through GC. This may increase possibility of false positive reports. Hence, in our implementation, we have excluded references from these special threads from the target of data race detection by default. This exclusion can be turned-off using an runtime option.

5 Experimental Results

In this section, we describe an experimental comparison of our read barrier based method and the write barrier based dynamic method. We used 64 bytes as sub-block size of an array for calculating the shared state.

[3]This modification of field reference includes accesses by undocumented `sun.misc.unsafe` class.

[4]There are other special threads, such as a reference handler thread that deals with weak references.

Application	Description	# threads
compress	Data compression program from SPEC JVM98	1
jess	Java expert shell system from SPEC JVM98	1
db	Memory resident database program from SPEC JVM98	1
javac	Java compiler program from SPEC JVM98	1
mpegaudio	MPEG audio decompression program from SPEC JVM98	1
mtrt	Multi-threaded ray-trace program from SPEC JVM98	3
jack	Java parser generator program from SPEC JVM98	1
SPECjbb	Java business benchmark program	42
Crypt	IDEA encryption program from Java Grande Benchmark	2
LUFact	LU factorization program from Java Grande Benchmark	2
SOR	Successive over-relaxation program from Java Grande Benchmark	2
Series	Fourier coefficient analysis program from Java Grande Benchmark	2
Sparse	Sparse matrix multiplication program from Java Grande Benchmark	2

Table 1: Benchmark Programs

Table 1 lists the benchmarks[14, 27] we used for the evaluation. The '# threads' column indicates the number of dynamically generated threads except system threads such as the finalizer thread. The Java Grande benchmarks are numerical benchmarks, and they perform many array operations on large shared array objects. Since they use the barrier based synchronization method, their manner of synchronization is not standard. However, they can reveal the overhead of race detection for array intensive applications.

Among these benchmarks, our system successfully found a data race on `RayTrace.threadCount` in mtrt benchmark. As reported in [2], this data race does not affect the correctness of the program execution.

All evaluations are performed using the HotSpot Java virtual machine ported to an HP-UX operating system. The environment of the experimental evaluation was HP9000/C3000(PA-8500 400MHz, 250MB main memory) with HP-UX11.0.

Figure 4 shows normalized execution time of the write barrier based method and the read barrier based method[5] compared to normal execution[6].

[5]The implementation of the write barrier based method is similar to the one for read barrier based method except that it recursively check object escape attributes at reference assignment instead of object reference.

[6]HotSpot Java VM optimizes execution of synchronization primitives for uncontended cases. However, since we need to maintain monitor stack at each monitor related bytecode execution, our read barrier and write barrier implementation does not use this optimization.

The write barrier based method requires a large execution overhead (69.7% to 3389.7% for the SPECjvm/jbb benchmarks, and 0.6% to 6777.0% for the Java Grande benchmarks) for detecting data races. In contrast, the read barrier based method can perform data race detection with a much lower overhead (57.8% to 735.7% for the SPECjvm/jbb benchmarks, and 0.6% to 6012.5% for the Java Grande benchmarks). We can see significant performance improvements of the read barrier based method over the write barrier based method on the mpegaudio and jack benchmarks. The performance improvements for these benchmarks indicate an increase in the number of dynamically examined objects because the write barrier based method deals with all globally reachable objects as targets of data race detection. The overheads for detecting data races for several of the Java Grande benchmarks are high compared with the SPEC benchmarks since these benchmarks frequently access shared arrays among threads with disjoint indexes.

To investigate the details of the field references, we obtained a breakdown of read/write references of the object fields. The result is shown in Figure 5. The WB and RB after each benchmark name indicate the write barrier based method and the read barrier based method, respectively. We can see from Figure 5 that the ratio of shared fields references of the write barrier based method are high for six SPEC benchmarks (compress, jess, javac, mpegaudio, jack, and SPECjbb) that have improved performance with the read barrier based method. In particular, nearly 40% of references are treated as targets of data race detection for the mpegaudio benchmark, for which the read barrier based method

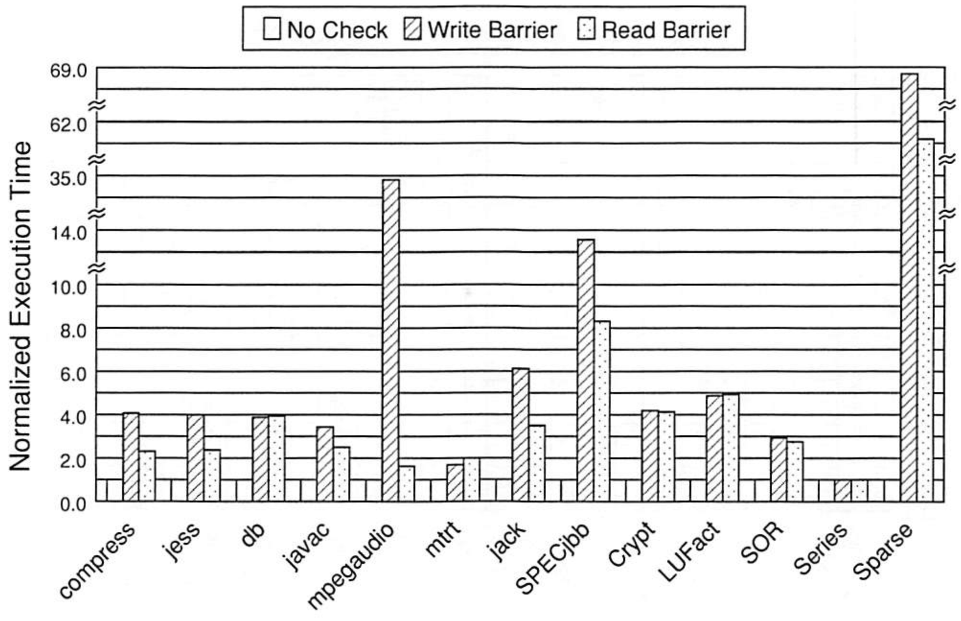


Figure 4: Execution time of the write barrier based method and the read barrier based method normalized to normal execution.

significantly improved performance.

Some SPEC benchmarks such as mtrt decreased the performance slightly by using the read barrier based method. We can see from Figure 5 that shared access ratio of these benchmarks does not decrease by the proposed method. This means that the performance differences arise from the read barrier overhead.

Compared with the SPEC benchmarks, the performance improvements for the Java Grande benchmarks are not as large. This is because the Java Grande benchmarks adopt a programming style that performs many accesses to small numbers of large shared arrays.

To confirm the effectiveness of array sub-block division, we compared the performance of benchmarks with and without array sub-block division. The result is shown in Figure 6. We can see that array sub-block division significantly improved performance of the SOR and Sparse benchmarks. It also made moderate improvements on the benchmarks such as SPECjbb, Crypt, or LUFact.

Division of arrays alone is effective for reducing overheads. For example, the shared field reference ratio of the write barrier based method without array division was almost 100% for mpegaudio, LUFact, and SOR benchmarks. Dividing arrays reduced this ratio to less than half, 1/117 for SOR, of the ratio without division.

Figure 7 shows the per object memory overhead of each race detection method including state field, monitor sets, monitor stacks, and data structures pointed to from the state field. The results of the write barrier based method do not include implicit memory overhead for object traversal on reference assignments. Note that the vertical axis of this graph is a logarithmic scale. The proposed method requires only a small number of bytes per object for the SPEC benchmarks. In contrast, the write barrier based method requires a large memory overhead for SPEC benchmarks such as compress and mpegaudio. Since the Java Grande benchmarks use large array objects, per object extra memory overhead is larger than with the SPEC benchmarks.

6 Conclusions and Future Work

In this paper, we proposed a novel data race detection method for object-oriented multi-threaded languages using dynamic escape analysis based on the read barrier. Data races are sources of errors

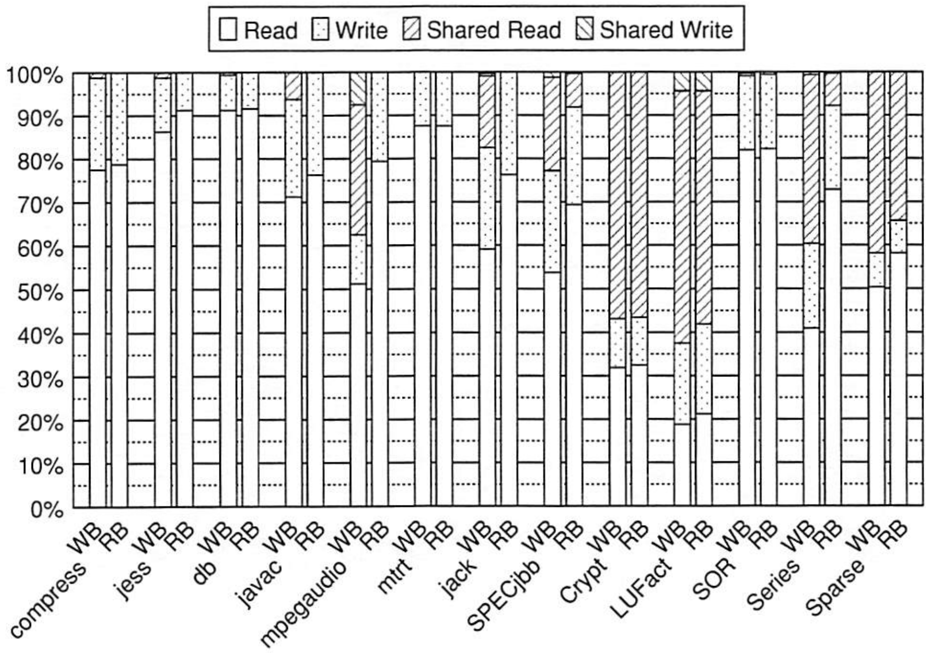


Figure 5: Breakdown of Object References

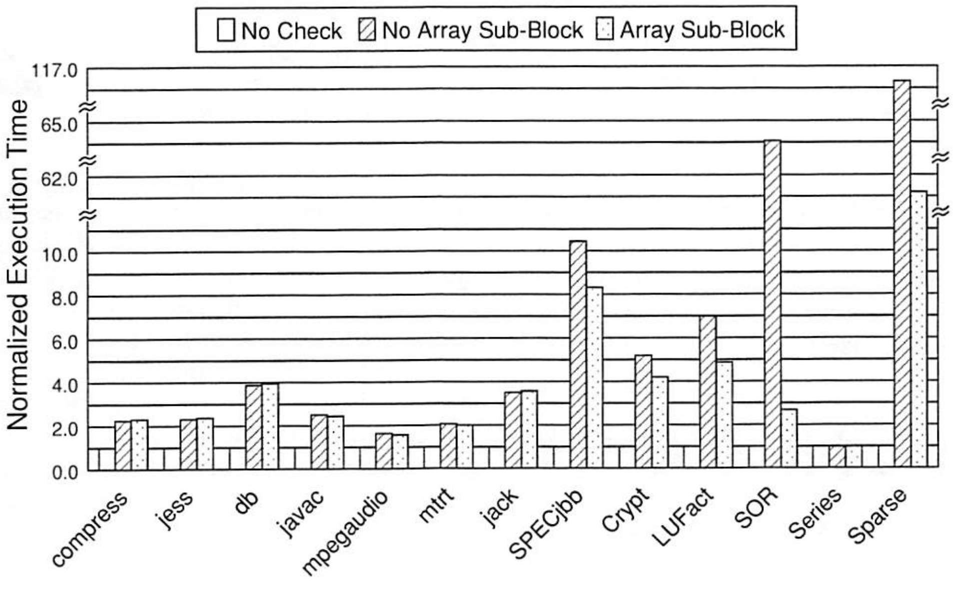


Figure 6: Relative Execution Time of Array Sub-Block Division

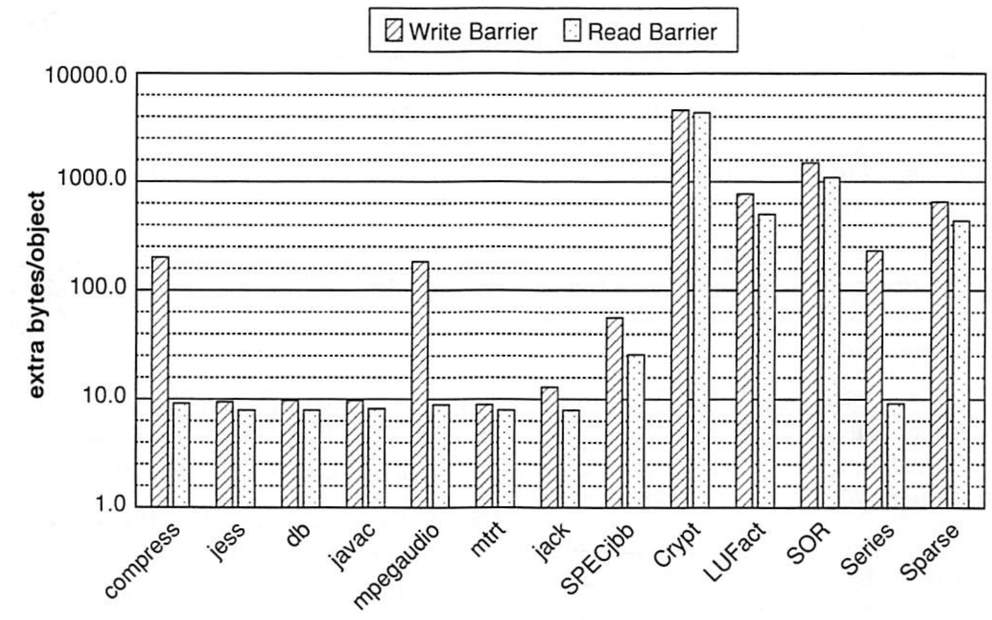


Figure 7: Extra Memory Overhead of Each Detection Method per Object

that are difficult to dissolve. Our method verifies whether a reference to an object is obtained by more than one thread at the point where reference fields are read out. By restricting the targets of data race detection to only escaped objects using the read barrier, we have successfully reduced data race detection overhead.

Our experimental results for a modified HotSpot Java virtual machine show that our method performs data race detection more effectively than the previous write barrier based method, and its detection overhead is small compared with normal execution. In addition to read barrier based filtering of objects, we developed a method that divides an array into sub-blocks as units of race detection. This division of arrays are also improves performance. Using these techniques, we can look for the occurrence of data races while incurring a relatively low overhead, 57.8% to 735.7% for SPEC benchmarks and 0.6% to 6012.5% for Java Grande benchmarks.

In the future, we intend to using static compiler optimizations for improving the execution performance. Static optimization such as common subexpression elimination can remove redundant checking of object/field status. Combination of static/dynamic escape analysis can also improve the performance by decreasing the number of objects that need dynamic checking.

Acknowledgments

We would like to thank the anonymous referees. Their comments were very useful for revising this paper.

References

[1] J.D. Choi. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[2] J.D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[3] J.D. Choi, B. Miller, and R. Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

[4] D.Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the SIGPLAN*

Conference on Programming Language Design and Implementation, pages 258–268, 1998.

[5] D. Dice. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[6] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991.

[7] J. Dolby and A. Chien. An Automatic Object Inlining Optimization and its Evaluation. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[8] T. Domani, G. Goldshtein, E.K. Kolodner, and E. Lewis. Thread-Local Heaps for Java. In *Proceedings of the 2002 Internationl Symposium on Memory Management*, 2002.

[9] M.B. Dwyer, J. Hatcliff, V.R. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. Technical Report SAnToS-TR2003-1, SAnToS Laboratory, Kansas State University, 2003.

[10] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[11] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *2000 International Conference on Compiler Construction*, 2000.

[12] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, 1992.

[13] J. Seward and N. Nethercote and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org/`, 2003.

[14] Java Grande Forum. Java Grande Forum Benchmark. `http://www.epcc.ed.ac.uk/javagrande/javag.html`, 2003.

[15] J.Bogda and U. Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 35–46, 1999.

[16] J.Gosling, B.Joy, and G.Steele. *The Java Language Specification.* Addison Wesley, 1999.

[17] KAI Software. Tutorial: Using Assure for Threads. `http://www.intel.com/software/products/assure/assuret_tutorial.pdf`, 2001.

[18] K.Rustan, M.Leino, and G. Nelson. An Extended Static Checker for Modula-3. In *Proceedings of 7th International Conference on Compiler Construction, LNCS1383*, 1998.

[19] L. Lamport. Time, clock, and the orderling of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[20] M.Christiaens and K.D.Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[21] Sun Microsystems. Java HotSpot Technology. `http://java.sun.com/products/hotspot/`, 2003.

[22] R. Netzer and B. Miller. What Are Race Condition? - Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), 1992.

[23] C.v. Praun and T.Gross. Object Race Detection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[24] R.Jones and R.Lins. *Garbage Collection - Algorighms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[25] E. Ruf. Effective Synchronization Removal for Java. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalarro, and T.E. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems*, 15(4):391, 411 1997.

[27] Standard Performance Evaluation Corp. SPEC benchmarks. `http://www.spec.org`, 2003.

[28] T.Lindholm and F.Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 2000.

Towards Dynamic Interprocedural Analysis in JVMs *

Feng Qian Laurie Hendren

School of Computer Science, McGill University
3480 University Street, Montreal, Quebec
Canada H3A 2A7
{fqian,hendren}@cs.mcgill.ca

Abstract

This paper presents a new, inexpensive, mechanism for constructing a complete call graph for Java programs at runtime, and provides an example of using the mechanism for implementing a dynamic reachability-based interprocedural analysis (IPA), namely dynamic XTA.

Reachability-based IPAs, such as points-to analysis and escape analysis, require a context-insensitive call graph of the analyzed program. Computing a call graph at runtime presents several challenges. First, the overhead must be low. Second, when implementing the mechanism for languages such as Java, both polymorphism and lazy class loading must be dealt with correctly and efficiently. We propose a new, low-cost, mechanism for constructing runtime call graphs in a JIT environment. The mechanism uses a profiling code stub to capture the first execution of a call edge, and adds at most one more instruction to repeated call edge invocations. Polymorphism and lazy class loading are handled transparently. The call graph is constructed incrementally, and it supports optimistic analysis and speculative optimizations with invalidations.

We also developed a dynamic, reachability-based type analysis, dynamic XTA, as an application of runtime call graphs. It also serves as an example of handling lazy class loading in dynamic IPAs.

The dynamic call graph construction algorithm and dynamic version of XTA have been implemented in Jikes RVM. We present empirical measurements of the overhead of call graph profiling and compare the characteristics of call graphs built using our profiling code stubs with conservative ones constructed by using dynamic class hierarchy analysis (CHA).

*The work was supported, in part, by NSERC.

1 Introduction

*Inter*procedural analyses (IPAs) derive more precise program information than *intra*procedural ones. Static IPAs provide a conservative approximation of runtime information to clients for optimizations. A foundation of IPA is the call graph of the analyzed program. IPAs for Object-Oriented (OO) programs share some common challenges. Virtual calls (polymorphism) make call graph construction difficult. Further, since the code base tends to be large, the complexity and precision of the analysis must be carefully balanced.

One difficulty of call graph construction for OO languages lies in how to approximate the targets of polymorphic calls. In addition to polymorphism, call graph construction for Java is further complicated by the presence of dynamic class loading. Static IPAs assume that the whole program is available at analysis time. However, this may not be the case for Java. A Java program can download a class file from the network or other unknown resources. Even when all programs exist on local disks, a VM may choose to load classes lazily, on demand, to reduce resource usage and improve responsiveness [21]. When a JIT compiler encounters an unresolved symbolic reference, it may choose to delay resolution until the instruction is executed at runtime. A dynamic analysis has to deal with these unresolved references. A more subtle problem, usually ignored by static IPAs for Java, is that a runtime type is defined by both the class name and its initial class loader. Therefore, a correct dynamic IPA has to be incremental (dealing with dynamic class loading), efficient, and type safe.

Although Java's dynamic features pose difficulties for program analyses, there are many opportunities at runtime that can only be enjoyed by dynamic analyses. For example, a dynamic analysis only needs to analyze loaded classes and invoked methods. Therefore, the analyzed code base can be much smaller than in a conservative static analysis. Further, dynamic class loading can improve the precision of type analyses. The set of runtime types can be limited to loaded classes. Thus, a dynamic analysis has more precise type information than its static counterpart. Further, in contrast to the conservative (pessimistic) nature of static analysis, a dynamic one can be optimistic about future execution, if used in conjunction with runtime invalidation mechanisms [12, 18, 23, 30].

Over the last 10 years, VM technology has greatly advanced. JIT compilers now implement most *intra*procedural data-flow analyses that can be found in static compilers [1, 22]. Further performance improvements have been achieved using adaptive and feedback-directed compilation [4, 5].

Dynamic *inter*procedural analysis, however, has not yet been widely adopted. Some type-based IPAs [18, 23] have gained ground in JIT compilation environments. However, work relating to more complicated, reachability-based IPAs, such as dynamic points-to analysis and escape analysis, is only just starting to emerge [15].

In this paper, we present a call graph construction mechanism for reachability-based interprocedural analyses at runtime. Instead of approximating a call graph as in static IPAs, our mechanism uses a profiling code stub to capture invoked call edges. The mechanism overcomes difficulties caused by dynamic class loading and polymorphism. Most overhead happens at JIT compilation and class loading time. It has only small overhead on the performance of applications in a JIT environment. A very desirable feature of the mechanism is that call graphs can be built incrementally while execution proceeds. This enables speculative optimizations using runtime invalidations for safety.

Dynamic IPAs seem more suitable for long-running applications in adaptive recompilation systems. Pechtchanski and Sarkar [23] described a general approach of using dynamic IPAs. A virtual machine gathers information about compiled methods and loaded classes in the initial state, and performs recompilation and optimizations only on selected hot methods. When the application reaches a "stable state", information changes should be rare.

Based on our new runtime call graph mechanism, we describe the design and implementation of an online version of an example IPA, XTA type analysis [32]. Dynamic XTA uses dependency databases to handle unresolved types and field references. The analysis is driven by VM events such as compilation, class loading, or the discovery of new call edges.

The rest of paper is organized as follows. Section 2 introduces our new call graph construction mechanism which serves as the basis for dynamic IPAs. In the following section, Section 3, we describe the design of a specific dynamic IPA, dynamic XTA type analysis, in the presence of lazy class loading. The call graph mechanism and dynamic XTA have been implemented in Jikes RVM. Section 4 analyzes the cost of call graph profiling and compares the characteristics of profiled call graphs to conservative ones built by dynamic CHA on a set of standard benchmarks. Section 5 discusses related work and conclusions are presented in Section 6.

2 Online Call Graph Construction

Context-insensitive call graphs are commonly used by IPAs, where a method is represented as one node in the call graph. There exists a directed edge from a method *A* to a method *B* if *A* calls *B*.

Dynamic class hierarchy information can be used to build a conservative call graph at runtime. However, it is desirable to have a more precise call graph for most interprocedural analyses. We propose a new mechanism for profiling and constructing context-insensitive call graphs at runtime. The mechanism initializes call edges using a profiling code stub. When the code stub gets executed, it generates a new call edge event, then it triggers method compilation if the method is not compiled yet, and finally patches the address of the real target. The mechanism captures the first execution event of each call edge, and the first execution has some profiling overhead. The repeated calls only need to execute at most one more instruction. Clients, such as call graph builders, can register callback routines called by a profiling code stub when new call edges are discovered. Callbacks can perform necessary actions before the callee is invoked.

The remainder of this section is structured as follows. First, in Section 2.1, we briefly introduce a conservative approach for building call graphs using runtime class hierarchy information. In Section 2.2 we give the necessary background, describing the existing implementation of virtual method tables in Jikes RVM. In Section 2.3 we describe the basic mechanism we propose for building call graphs at runtime, and in Section 2.4 we show how this basic mechanism can be optimized to reduce overheads.

2.1 Conservative call graph construction using dynamic CHA

Due to polymorphism, the exact types of the receiver of a virtual call site may not be known at analysis time. Class hierarchy analysis (CHA) [9] makes the conservative assumption that all subtypes of a receiver's declaring type are possible types at runtime.

CHA was originally suggested as a static analysis, where all classes and the complete class hierarchy are known at compile time. However, when adapting CHA to be a dynamic analysis one must consider that the class hierarchy can grow as classes are dynamically loaded. Thus a dynamic CHA must record all virtual call sites that have already been resolved. When a new class is loaded, it must be included in the the type set of any recorded call site whose receiver's declaring class is a super type of the newly loaded class. If the newly added type, of a call site, declares a method with the same signature as the callee, a new call edge to the method must be generated at this call site. Hirzel et. al. [15] have given a detailed description of this approach. In our study, we implemented a call graph builder using dynamic CHA to compare with our proposed profiler-based mechanism.

2.2 Background: virtual method table

We propose a profiling mechanism for constructing a more precise dynamic call graph than a conservative one constructed using dynamic CHA. To understand how the mechanism works, we first revisit the virtual method dispatch table in Jikes RVM [1], which is a standard implementation in modern Java virtual machines. Figure 1(a) depicts the object layout in Jikes RVM. Each object has a pointer, in its header, to the Type Information Block (TIB) of its type (class). A TIB is an array of objects that encodes the type information of a class. At a fixed offset from the TIB header is the Virtual Method Table (VMT) which is embedded in the TIB array. A resolved method has an entry in the VMT of its declaring class, and the entry offset to the TIB header is a constant, say `method_offset`, assigned during class resolution. A VMT entry records the instruction address of the method that owns it. Figure 1(b) shows that, if a class, say `A`, inherits a method from its superclass, `java.lang.Object`, the entry at the method offset in the subclass' TIB has the inherited method's instruction address. If a method in the subclass, say `D`, overrides a method from the superclass, the two methods still have the same offset, but the entries in two TIBs point to different method instructions.

Given an object pointer at runtime, an *invokevirtual* bytecode is implemented by three basic operations:

```
TIB = * (ptr + TIB_OFFSET);
INSTR = TIB[method_offset];
JMP INSTR
```

The first instruction obtains the TIB address from the object header. The address of the real target is loaded at the `method_offset` offset in the TIB. Finally the execution is transferred to the target address.

Lazy method compilation works by first initializing TIB entries with the address of a lazy compilation code stub. When a method is invoked for the first time, the code stub gets executed. The code stub triggers the compilation of the target method and patches the address of the compiled method into the TIB entry (where the code stub resided before).

2.3 Call graph construction by profiling

In normal lazy method compilation, the code stub captures the first invocation of a method without distinguishing callers. In order to capture call edges, we extended the TIB structure to store information per caller. Figure 1(c) shows the extended TIB structure. The TIB entry of a method is replaced by an array of instruction addresses. We call the array a Caller-Target Block (CTB). The indices of CTB slots (*caller_index*) are dynamically assigned to callers of the method by the JIT compilers. Note that now an *invokevirtual* bytecode takes one extra load to get the target address.

```
TIB = * (ptr + TIB_OFFSET);
/* load method's CTB array from TIB */
CTB = TIB[method_offset];
/* load method's code address */
INSTR = CTB[caller_index];
JMP INSTR
```

The lazy method compilation code stub is extended to a profiling code stub which, in addition to triggering the lazy compilation of the callee, also generates a new call edge event from the caller to the callee. Initially all of the CTB entries have the address of the profiling code stub. When the code stub at a CTB entry gets executed, it notifies clients monitoring new call edge events, and compiles the callee method if necessary. Finally the code stub patches the callee's instruction address into the CTB entry. Clearly the profiling code stub at each entry of the CTB array will execute at most once, and the rest of the invocations from the same caller will execute the callee's machine instruction directly.

There remain four problems to address. First, one needs a convenient way of indexing into the CTBs which works even in the presence of lazy class loading. Second, the implementation of interface calls should be aware of the CTB array. Third, object initializers and static methods can be handled specially. Fourth, we must handle the case where an optimizing compiler inlines one method into another. Our solution to these four problems is given below.

2.3.1 Allocating slots in the CTB

To index callers of a callee, our modified JIT compiler maintains a table of *(callee, caller)* pairs. In Java bytecode, the target of *invokevirtual* is only a symbolic reference to the name and descriptor of the method as well as a symbolic reference to the class where the method can be found. Resolving the method reference requires the class to be loaded first. A VM can delay method resolution until the call instruction is executed at runtime.

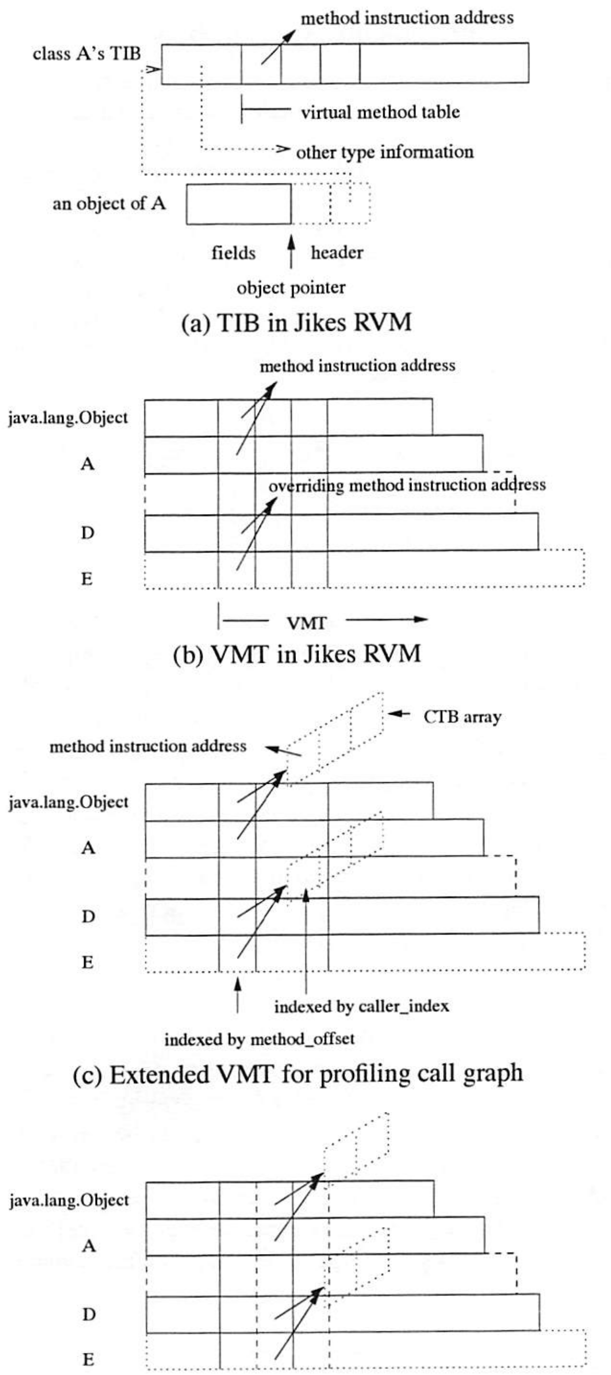


Figure 1: Virtual Method Dispatching Table in Jikes RVM

Therefore, the real target may not be known at JIT compilation time.

To deal with lazy class resolution and polymorphism, our approach uses the callee's method name and descriptor in the table. For example, if both methods `X.x()` and `Y.y()` have virtual calls of a symbolic reference `A.m()`, and another method `Z.z()` has a virtual call of `B.m()`, our approach assumes that all three methods are possible callers of any method with the signature `m()`[1], and allocates slots in the TIB for all of them. At runtime, only two CTB entries of `A.m()` may be filled, and only one entry of `B.m()` may get filled. With this solution no accuracy is lost, but some space may be wasted due to unfilled CTB entries. Although some space is sacrificed, our approach simplifies the task of handling symbolic references and polymorphism. In real applications we observed that only a few common method signatures, such as `equals(java.lang.Object)`, and `hashCode()`, have large caller sets where space is unused.

2.3.2 Approximating interface calls

Interface calls are considered to be more expensive than virtual calls in Java programs because a normal class can only have a single super class, but could implement multiple interfaces. Jikes RVM has an efficient implementation of interface calls using a interface method table with conflict resolution stubs [2].

We tried two approaches to handling interface calls in the presence of CTB arrays. Our first approach profiles interface calls by allocating a *caller_index* for a call site in the JIT compiler and generating an instruction before the call to save the index value in a known memory location. After a conflict resolution stub has found its target method, it loads the index value from the known memory location. The CTB array of the target method is loaded from the TIB array of receiver object's declaring class. The target address is read out from the CTB at the index, and finally the resolution stub jumps to the target address. This approach uses two more instructions to store and load the *caller_index* than *invokevirtual* calls. After introducing one of our optimizations in Section 2.4, inlining CTB elements into TIBs, the conflict resolution stub requires more instructions to check the range of the index value to determine if the indexed CTB element is inlined in the TIB or not.

Our second approach was to simply use dynamic CHA to build call edges for *invokeinterface* call sites, without introducing profiling instructions.

Since our profiling results showed that the number of

[1]A full method descriptor should include the name of the method, parameter types, and the return type. In this example, we use the name and parameter types only for simplicity.

call edges from *invokeinterface* call sites is only a small portion of all edges, we chose to use the second approach for the remaining experiments in this paper.

2.3.3 Handling object initializers and static methods

Because there are many object initializers that share a common name `<init>` and descriptor, their CTB arrays may grow too large if we allocate CTB slots using the name and descriptor as index. Since calls of object initializers and static methods are monomorphic, the allocation of CTB slots for each method is independent of other methods with the same name and descriptor. For example, static methods `A.m()` and `B.m()` both can use the same CTB index for different callers. Therefore, there is no superfluous space in CTB arrays of object initializers and static methods. For unresolved static or object initializer method references, a dependency on the reference from the caller is registered in a database. When the method reference gets resolved (this happens due to a class loading event), the dependency is converted to a call edge conservatively. Using the *_213_javac* benchmark as example, we found this conservativeness only adds 1.5% more edges.

2.3.4 Dealing with Inlining

In an adaptive system, inlining might be applied on a few hot methods. We capture these events as follows. When a callee is inlined into a caller by an optimizing JIT compiler, the call edge from the caller to callee is added to the call graph unconditionally. This is a conservative solution without runtime overhead. Since an inlined call site is likely executed before its caller becomes hot, the number of added superfluous edges is modest.

2.4 Optimizations

Since Jikes RVM is written in Java, our runtime call graph construction mechanism may incur two kinds of overhead. First, adding one instruction per call can potentially consume many CPU cycles because Jikes RVM itself is compiled using the same compilers used for compiling the applications, and it also inserts many system calls into applications for runtime checks, locks and object allocations. Second, a CTB array is a normal Java array with a three-word header; thus CTB arrays can increase memory usage and create extra work for garbage collectors.

Table 1 shows the distribution of the CTB sizes for the SpecJVM98 benchmarks [27] profiled in a *FastAdaptiveSemiSpace* boot image. The boot image contains mostly RVM classes and a few Java utility classes. We only profiled methods from Java libraries and benchmarks. A small number of methods of classes in the boot image may have CTB arrays allocated at runtime because there is no clear cut mechanism for distinguishing between Jikes RVM code and application code. The first column shows the range of the number of callers. The second and third columns list the distributions of methods belonging to Java libraries and SpecJVM application code.[2] To demonstrate that most methods have few callers, we calculated the cumulative percentages of methods that have no caller, ≤ 1, ≤ 3 and ≤ 7 callers in the first to fourth rows. We found that 89% of methods from (loaded classes in) Java libraries and 84% of methods from SpecJVM98 have no more than 7 callers. In these cases, it is not wise to create short CTB arrays because each array header takes 3 words. The last data row labelled "`TOTAL`" gives the total number of methods of all classes and the number of methods in each of two subcategories.

#callers	Java Libraries		SpecJVM App	
0	2214	69.08%	507	19.32%
1	291	78.16%	815	50.38%
2-3	172	83.53%	608	73.55%
4-7	170	88.83%	283	84.34%
8-	358		411	
TOTAL	3205		2624	

Table 1: Distribution of CTB sizes

To avoid the overhead of array headers for CTBs, and to eliminate the extra instruction to load the CTB array from a TIB in the code for *invokevirtual* instructions, a local optimization is to inline the first few elements of the CTB into the TIB. Since caller indices are assigned at compile time, a compiler knows which part of the CTB will be accessed in the generated code. To accommodate the inlined part of the CTB, a class' TIB entry is expanded to allow a method to have several entries. Figure 1(d) shows the layout of TIBs with one inlined CTB element. When generating instructions for a virtual call, the value of the caller's CTB index, `caller_index`, is examined: if the index falls into the inlined part of the CTB, then invocation is done by three instructions:

```
TIB = * (ptr + TIB_OFFSET);
INSTR = TIB[method_offset + caller_index];
JMP INSTR
```

Whenever a CTB index is greater than or equal to the inlined CTB size, *INLINED_CTB_SIZE*, then four instructions must be used for the call:

```
TIB = * (ptr + TIB_OFFSET);
CTB = TIB[method_offset + CTB_ARRAY_OFFSET]
INSTR = CTB[caller_index - INLINED_CTB_SIZE
JMP INSTR
```

[2] We used package names to distinguish classes.

Note that in addition to saving the extra instruction for inlined CTB entries, the space overhead of the CTB header is eliminated in the common cases where all CTB entries are inlined.

Another source of optimization is to avoid the overhead of handling system code, such as runtime checks and locks, inserted by compilers, because this code is frequently called and ignoring them does not affect the semantics of applications. To achieve this, the first CTB entry is reserved for the purpose of system inserted calls. Instead of being initialized with the address of a call graph profiling stub, the first entry has the address of a lazy method compilation code stub or method instructions. When the compiler generates code for a system call, it always assigns the *zero* `caller_index` to the caller. To avoid the extra load instruction, the first entry of a CTB array is always inlined into the TIB.

3 Dynamic XTA Type Analysis

A runtime call graph is constructed incrementally while a program runs. Dynamic IPAs using call graphs also have to perform analysis incrementally. A dynamic IPA has to overcome the difficulties of dynamic class loading and lazy resolution of references. As one example application of profiled call graphs, we developed a dynamic XTA type analysis which can serve as a general model of other dynamic IPAs.

Tip and Palsberg [32] proposed a set of propagation-based call graph construction algorithms for Java with different granularities ranging from RTA to 0-CFA. XTA uses separate sets for methods and fields. A type reaching a caller can reach a callee if it is a subclass of the callee's parameter types. Types can be passed between methods by field accesses as well. To approximate the targets of a virtual call, XTA uses the reachable types of the caller to perform method lookups statically. When new targets are discovered, new edges are added into the graph. The analysis performs propagation until reaching the fixed point. XTA has the same complexity as subset-based points-to analysis, $O(n^3)$, but with fewer nodes in the graph.

XTA analysis is a good candidate as a dynamic IPA: it requires reasonably small resources to represent the graph since it ignores the dataflow inside a method. The results might be less precise than an analysis using a full dataflow approach [23, 31]. On the other hand, rich runtime type information may improve the precision of dynamic XTA. The results of the analysis can be used for method inlining and the elimination of type checks.

Like other static IPAs for Java, static XTA assumes the whole programs are available at analysis time. Dynamically-loaded classes must be supplied to the analysis manually. The burden on static XTA is to approximate targets of polymorphic calls while propagating types along the call graph. However, dynamic XTA does not have this difficulty because it uses the call graph constructed at runtime. The call graph used by dynamic XTA is significantly smaller than the one constructed by static XTA. However, a new challenge for dynamic XTA comes from lazy class loading. In a Java class file, a call instruction has only the name and descriptor of a callee as well as a symbolic reference to a class where the callee can be found. Similarly, field access instructions have symbolic references only. At runtime, a type reference is resolved to a class type and a method/field reference is resolved to a method/field before any use. [3]

```
class A { Object f; }

class B extends A {
}

  A a;
  a.f = ...;

  B b;
  o = b.f;
```

(a) Java source

```
......
putfield A.f Ljava/lang/Object;

getfield B.f Ljava/lang/Object;
......
```

(b) compiled bytecode

Figure 2: Field reference example

Figure 2 shows a simple example to help understand the problem caused by symbolic references. Class *B* extends class *A*, which declares a field *f*. Field accesses of *a.f* and *b.f* were compiled to *putfield* and *getfield* instructions with different symbolic field references *A.f* and *B.f*. At runtime, before the *getfield* instruction gets executed, the reference *B.f* is resolved to field *f* of class *A*. However, a dynamic analysis or compiler cannot determine that *B.f* will be resolved to *f* of *A* without loading both classes *B* and *A*.

Resolution of method/field references requires the classes to be loaded and resolved first. However, a JVM may choose to delay such resolution as late as possible to reduce resource usage and improve responsiveness [21]. To port a static IPA for Java to a dynamic IPA, the analysis must be modified to handle unresolved references.

[3] In following presentation, we use type(s) as a short name for resolved class type(s), and use references for symbolic references, e.g., type references, method/field references.

In this section, we demonstrate a solution for the problem for dynamic XTA; our solution is also applicable to general IPAs for Java.

Our dynamic XTA analysis constructs a directed XTA graph $G = \{V, E, TypeFilters, ReachableTypes\}$:

- $V \subseteq M \cup F \cup \{\alpha\}$, where M is a set of resolved methods, F is a set of resolved fields, and α is an abstract name representing array elements;
- $E \subseteq V \times V$, is the set of directed edges;
- $TypeFilters \subseteq E \rightarrow S$, is a map from an edge to a set of types, S;
- $ReachableTypes \subseteq V \rightarrow T$, is a map from a node to a set of resolved types T.

The XTA graph combines call graphs and field/array accesses. A call from a method *A* to a method *B* is modelled by an edge from node *A* to node *B*. The filter set includes parameter types of method *B*. If *B*'s return type is a reference type, it is added in the filter set of the edge from *B* to *A*. Field reads and writes are modelled by edges between methods and fields, with the fields' declaring classes in the filter. Each node has a set of reachable (resolved) types.

Basic graph operations include adding new edges, new reachable types, and propagations:

- *addEdge(a, b, T)*, creates an edge from a node *a* to a *b* if it does not exist yet; then adds types in set T to the filter set associated with the edge;
- *addType(a, t)*, adds a type *t* to the reachable type set of a node *a*;
- *propagate(t, a)*, propagates a type *t* to successors of a node *a* if *t* is a subtype of a type in the filter set associated with the edge from *a* to its successor. If *t* is not in a successor's reachable type set, it is recursively propagated to all of that successor's descendants until there are no further changes.

Since the call graph is constructed at runtime using code stubs, there are no new edges created during propagation. The complexity of the propagation operation is linear.

Dynamic XTA analysis is driven by events from JIT compilers and class loaders. Figure 3 shows the flow of events. In the dotted box are the three modules of dynamic XTA analysis: XTA graphs, the analysis, and dependency databases. The JIT compilers notify the analysis by channel 1 that a method is about to be compiled. The analysis scans the bytecode of the method body and, for each *new* instruction with a resolved type, the analysis adds the type into the reachable type set of the method via channel 3; otherwise it registers a dependency on the unresolved type reference for the method via channel 4. Similarly for field accesses, if the field reference can be resolved without triggering class loading, the analysis adds a directed edge into the graph via channel 3; otherwise, it registers a dependency on unresolved field reference for the method. Since we use call graph profiling code stubs to discover new call edges, the code stubs can add new edges to the graph by channel 2. Whenever a type reference or field reference gets resolved, the dependency databases are notified (by channel 5), and registered dependencies on resolved references are resolved to new reachable types or new edges of the graph. Whenever the graph is changed (either an edge is changed, or a new reachable type is added), a propagator propagates type sets of related nodes until no further change occurs.

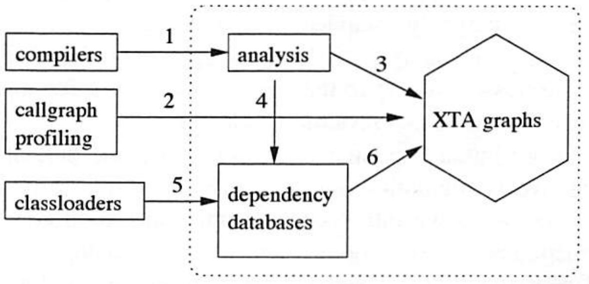


Figure 3: Model of XTA events

Compared to static IPAs such as points-to analysis, the problem set of dynamic XTA analysis is much smaller because the graph contains only compiled methods and resolved fields at runtime. Although optimizations such as off-line variable substitution [24] and online cycle elimination [11] can help reduce the graph size further, the complexity and runtime overhead of the algorithms may prevent them from being useful in a dynamic analysis. Efficient representations for sets and graphs, such as hybrid integer sets [14, 20] and BDDs [7], are more important since the dynamic analysis has bounded resources. In our implementation, graphs, sets, and dependency databases were implemented using hybrid integer sets and integer hash maps.

Graph changes are driven by runtime events such as newly compiled methods, newly discovered call edges, or dynamically loaded classes. Similar to the DOIT framework [23], clients using XTA analysis for optimizations should register properties to be verified when the graph changes. Since the analysis can notify the client when a change occurs, the clients can perform an invalidation of compiled code or recover execution states to a safe point. The exact design and implementation details for verifying properties and performing invalidations are beyond the scope of this paper. Readers can find more about dependency management and invalidation techniques in [12, 16, 18].

4 Evaluation

We have implemented our proposed call graph construction mechanism in Jikes RVM [19] v2.3.0. Our benchmark set includes the SpecJVM98 suite [27], SpecJBB2000 [26], and a CFS subset evaluator from a data mining package Weka [34]. We made a variation of the *FastAdaptiveCopyMS* boot image for evaluating our mechanism. In our experiment, classes whose names start with `com.ibm.JikesRVM` are not presented in the dynamic call graphs because (1) the number of RVM classes is much larger than the number of classes of applications and libraries, and (2) the classes in the boot image were statically compiled and optimized. Static IPAs such as extant analysis [28] may be applied on the boot image classes. We report the experimental results for application classes and Java library classes.

In our initial experiments we found that the default adaptive configuration gave significantly different behaviour when we introduced dynamic call graph construction because the compilation rates and speedup rates of compilers were affected by our call graph profiling mechanism. It was possible to retrain the adaptive system to work well with our call graph construction enabled, but it was difficult to distinguish performance differences due to changes in the adaptive behaviour from differences due to overhead from our call graph constructor. In order to provide comparable runs in our experiments, we used a counter-based recompilation strategy and disabled background recompilation. We also disabled adaptive inlining. This configuration is more deterministic between runs as compared to the default adaptive configuration. This behavior is confirmed by our observation that, between different runs, the number of methods compiled by each compiler is very stable. The experiment was conducted on a PC with a 1.8G Hz Pentium 4 CPU and 1G memory. The heap size of RVM was set to 400M. Note that Jikes RVM and applications share the same heap space at runtime.

The first column of Table 2 gives four configurations of different inlined CTB sizes and the default *FastAdaptiveCopyMS* configuration without the dynamic call graph builder. The boot image size was increased about 10%, as shown in column 2, when including all compiled code for call graph construction. Inlining CTB elements increases the size of TIBs. However, changes are relatively small (the difference between inlined CTB sizes 1 and 2 is about 153 kilobytes), as shown in the second column.

The third column shows the memory overhead, in bytes, of allocated CTB arrays for methods of classes in Java libraries and benchmarks when running the *_213_javac* benchmark with an input size 100. The time for creating, expanding and updating CTB array is negligible.

Inlined CTB sizes	bootimage size (bytes)		CTB space (bytes)
default	24,477,688	N/A	N/A
1	26,915,236	9.96%	678,344
2	27,068,340	10.58%	660,960
4	27,327,760	11.64%	637,312
8	27,838,796	13.73%	607,712

Table 2: Bootimage sizes and allocated CTB sizes of _213_javac

A Jikes RVM-specific problem is that the RVM system and applications share the same heap space. Expanding TIBs and creating CTBs consumes heap space, leaving less space for the applications, and also adding more work for the garbage collectors. We examine the impact of CTB arrays on the GC. Since CTB arrays are likely to live for a long time, garbage collection can be directly affected. Using the *_213_javac* benchmark as example with the same experimental setting mentioned before, GC time was profiled and plotted in Figure 4 for the default system and configurations with different inlined CTB sizes. The x-axis is the garbage collection number during the benchmark run, and the y-axis is the time spent on each collection. We found that, with these CTB arrays, the GC is slightly slower than the default system, but not significantly. When inlining more CTB elements, the GC time is slightly increased. This might be because the increased size of TIBs exceeds the savings on CTB array headers when the inlining size gets larger. We expect a VM with a specific system heap would solve this problem.

The problem mentioned above also poses a challenge for measuring the overhead of call graph profiling. Furthermore, the call graph profiler and data structures are written in Java, which implies execution overhead and memory consumption, affecting benchmark execution times. To only measure just the overhead of executing profiling code stubs, we used a compiler option to replace the allocated caller index by the *zero* index. When this option is enabled, calls do not execute the extra load instruction and profiling code stub, but still allocate CTB arrays for methods. For CFS and SpecJVM98 benchmarks, we found that usually the first run has some performance degradation when executing profiling

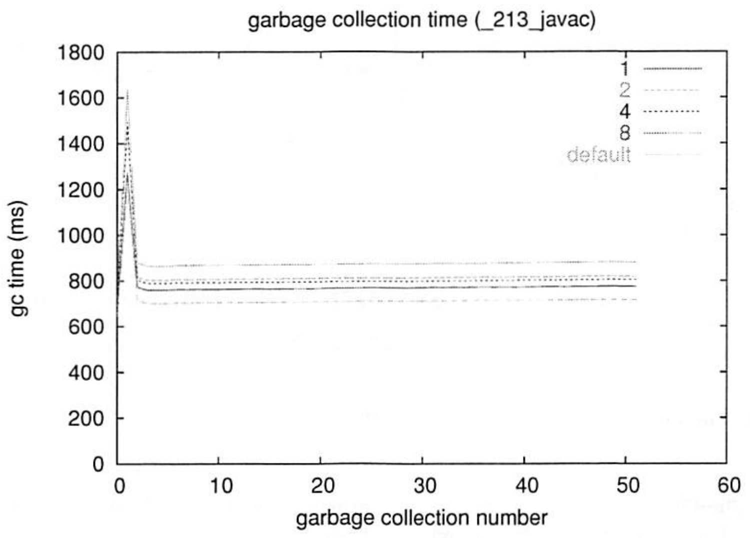


Figure 4: GC time when running _213_javac

code stubs (up to 9% except for _201_compress[4]), but the degradation is not significant upon reaching a stable state (between -2% to 3%). The performance of SpecJBB2000 is largely unaffected. Compared to not allocating CTB arrays at all (TIBs, however, are still expanded), the performance change is also very small. For our set of benchmarks, it seems that inlining more CTB array elements does not result in further performance improvements.

Table 3 shows the size of the profiled call graph compared to the one constructed by using dynamic CHA. The size of the call graph generated by CHA is shown in the second and third columns where, in the second column, the total number of call edges is followed by those from *invokevirtual* call sites only. The number of methods is given in the third column. From the second column, we can see that for all benchmarks, the major part of call edges come from *invokevirtual* instructions. The fourth and fifth columns show the size of profiled call graph and the percentages comparing to the sizes of the CHA call graphs. Call graphs constructed using our profiling code stubs have 20% to 50% fewer call edges than the CHA-based ones. More call edges from *invokevirtual* sites were reduced than for the other types of call instructions because we took the conservative CHA approach on other types of call instructions to reduce runtime overhead. The reduction for the number of methods is not as significant as for the number of call edges.

[4]The first run of _201_compress does not promote enough methods to higher optimization levels.

5 Related Work

Static call graph construction for OO programming languages focuses on approximating a set of types that a receiver of a polymorphic call site may have at runtime. Static class hierarchy analysis (CHA) [9] treats all subclasses of a receiver's declaring class as possible types at runtime. Rapid type analysis (RTA) [6] prunes the type set of CHA by eliminating types that do not have an allocation site in the whole program. Static CHA and RTA examine the entire set of classes. Propagation-based algorithms propagate types from allocation sites to receivers of polymorphic call sites along a program's control flow. Assignments, method calls, field and array accesses may pass types from one variable to another. Context-insensitive algorithms can be modelled as unification-based [29] or subset-based [3] propagation as points-to analysis. The complexity varies from $O(N\alpha(N,N))$ for unification-based analysis to $O(N^3)$ for subset-based analysis. Context-sensitive algorithms [10] might yield more precise results but are difficult to scale to large programs. Since CHA and RTA do not use control flow information, both are considered to be fast algorithms when compared with propagation-based algorithms. Both VTA [31] and XTA analysis [32] are simple propagation-based type analyses for Java. The analyses can either use a call graph built by CHA/RTA, then refine it, or build the call graph on the fly [25].

Ishizaki et. al. [18] published a new method of utilizing class hierarchy analysis for devirtualization at runtime. If the target of a virtual call is not overridden in the current class hierarchy, a compiler may choose to inline the target directly with a backup code of normal vir-

benchmark	CHA			Profiling		
	#edges		#methods	#edges		#methods
compress	733	458	365	516 (70%)	241 (53%)	303 (83%)
jess	2549	1364	1130	1986 (78%)	801 (59%)	802 (71%)
db	961	578	413	711 (74%)	328 (57%)	350 (84%)
javac	9427	8137	1662	4315 (46%)	3025 (37%)	1169 (70%)
mpegaudio	1228	849	645	853 (69%)	475 (56%)	474 (73%)
mtrt	1192	833	563	950 (80%)	591 (71%)	446 (79%)
jack	1746	1131	703	1413 (81%)	799 (71%)	572 (81%)
jbb	4166	2802	1394	3221 (77%)	1757 (63%)	1160 (83%)
CFS	2101	1552	843	1259 (60%)	712 (46%)	557 (66%)

Table 3: The number of call edges and methods discovered by CHA and Profiling

tual call. To cope with dynamic class loading, the runtime system monitors class loading events. If a newly loaded class overrides a method that has been directly inlined in some callers, the code of callers has to be patched with the backup path before class loading proceeds. Pechtchanski and Sarkar [23] presented a framework for performing dynamic optimistic interprocedural analysis in a Java virtual machine. Similar to dynamic CHA, their framework builds detailed dependencies between optimistic assumptions for optimizations and runtime events such as method compilation. Invalidation is a necessary technique for correctness when the assumption is invalidated. Neither of these approaches explored reachability-based analysis which requires a call graph as the base. Our work inherits the merits of their work, supporting optimistic optimizations and invalidations. Bogda and Singh [8] experimented an online interprocedural shape analysis, which uses an inlining cache to construct the call graph at runtime. However, their implementation was based on bytecode instrumentation, which incurs a large overhead. Our work aims to build an accurate call graph with little overhead to enable reachability-based IPAs at runtime.

In parallel to our work, Hirzel et. al. [15] adapted a static subset-based pointer analysis to a runtime analysis in Jikes RVM. In their work, an approach similar to ours is used to handle lazy class loading and unresolved method references. However, they used a conservative call graph constructed by dynamic CHA, and the analysis also considers the dataflow in a method. It would be interesting to see how much the smaller call graph produced by our mechanism could improve their pointer analysis results.

Code-patching [18] and stack-rewriting [12, 17] are necessary invalidation techniques for optimistic optimizations. Those operations might be expensive at runtime. An optimization client should use these techniques wisely. For example, if an optimistic optimization has rare invalidations, these techniques can be applied. In situations of frequent invalidations or incomplete IPA information, an optimization may choose runtime checks to guard optimized code.

Static IPAs for Java programs assume whole classes are available at analysis time. Dynamically loaded classes should be supplied to the analysis manually. Sreedhar et.al. [28] proposed an *extant analysis framework* which performs unconditional static optimizations on references that can only have types in the closed world (known classes by analysis), and guided optimizations on references with possible dynamically loaded types. However, the effectiveness of online *extant analysis* may be compromised by the laziness of class loading at runtime. Java poses access restrictions on fields by modifiers. Field analysis [13] uses access modifiers to derive useful properties of fields for optimizations.

A new wave of VM technology is adaptive feedback-directed optimizations [4, 16, 22]. Sampling is a technique for collecting runtime information with low costs. Profiling information provides advice to compilers to allocate resources for optimizing important code areas. Compared with feedback-directed optimizations, optimizations based on dynamic IPAs can be optimistic using invalidation techniques instead of using runtime checks. Dynamic IPAs also provide a complete picture of an executing program. The new proposed mechanism is capable of finding all invoked call edges in executed code. In many cases, profiling information can be aggregated with IPAs. For example, Jikes RVM's adaptive system samples call stacks periodically and builds a weighted, partial call graph for adaptive inlining. A complete call graph constructed by our mechanism could be annotated with sampled weights on edges and clients could perform probabilistic analysis using the call graph.

6 Conclusions

In this paper we have proposed a new runtime call graph construction mechanism for dynamic IPAs in a JIT environment. Our approach uses code stubs to capture the first-time execution of a call edge. The new mechanism avoids iterative propagation which is costly at runtime. We also addressed another important problem faced by dynamic IPAs: lazy class loading. Our approach handles the problem transparently. An important characteristic of our mechanism is that it supports speculative optimizations with invalidation backups. Our preliminary results showed that the overhead of online call graph construction is very small, and the call graph is much smaller than the one built by dynamic CHA.

Based on runtime call graphs, we outlined the design of a dynamic XTA type analysis. The model of handling unresolved references is applicable to other dynamic IPAs.

Based on the encouraging results so far, we are working on combining call graph profiling and dynamic CHA to deal with boot images and JNI calls. We also plan to use the results of dynamic XTA to expose more opportunities for method inlining. We are also planning to use the runtime call graphs, and the fundamental approach already used for dynamic XTA, for developing other dynamic reachability-based IPAs, e.g. escape analysis [33].

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] B. Alpern, A. Cocchi, S. J. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, 2001.

[3] L. O. Andersen. Program Analysis and Specialization for the C Programming Language, May 1994. Ph.D thesis, DIKU, University of Copenhagen.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adapative Optimization in the Jalapeño JVM. In *Proceedings the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Oct 2000.

[5] M. Arnold, M. Hind, and B. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 111 – 129, October 2002.

[6] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324 – 341, Oct 1996.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to Analysis Using BDDs. In *Proceedings of the Conference on Programming Language Design and Inplementation*, pages 103–114, June 2003.

[8] J. Bogda and A. Singh. Can a Shape Analysis Work at Run-time? In *USENIX Java Virtual Machine and Technology Symposium*, pages 13 – 26, April 2001.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *ECOOP'95—9th European Conference for Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, August 1995. Springer.

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[11] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 85–96, June 1998.

[12] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization*, pages 241 – 252, March 2003.

[13] S. Ghemawat, K. Randall, and D. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 334 – 344, June 2000.

[14] N. Heintze. Analysis of Large Code Bases: The Compile-Link-Analyze Model, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`.

[15] M. Hirzel, A. Diwan, and M. Hind. Pointer Analysis in the Presence of Dynamic Class Loading. In *ECOOP'04 – 18th European Conference for Object-Oriented Programming*, June 2004.

[16] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming, 1994. Ph.D Thesis, Standford University.

[17] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the Conference on Programming Language Design and Implementations*, pages 32 – 43, July 1992.

[18] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 294–310, October 2000.

[19] Jikes^{TM} Research Virtual Machine. `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.

[20] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[21] S. Liang and G. Bracha. Dynamic Class Loading in the Java(TM) Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36 – 44, October 1998.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1 – 12, April 2001.

[23] I. Pechtchanski and V. Sarkar. Dynamic Optimistic Interprocedural Analysis: A Framework and an Application. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 195 – 210, October 2001.

[24] A. Rountev and S. Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 47 – 56, June 2000.

[25] A. Rountev, A. Milanova, and B. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43 – 55, October 2001.

[26] Spec JBB2000 benchmark. `http://www.spec.org/jbb2000/`.

[27] Spec JVM98 benchmarks. `http://www.spec.org/jvm98/`.

[28] V. C. Sreedhar, M. G. Burke, and J.-D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the Conference on Programming Language Design and Implementations*, pages 196 – 207, June 2000.

[29] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[30] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 312 – 323, June 2003.

[31] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 264–280, October 2000.

[32] F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, October 2000.

[33] F. Vivien and M. C. Rinard. Incrementalized Pointer and Escape Analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 35 – 46, May 2001.

[34] Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`.

Java™ Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications

Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan
Toronto Lab, IBM Canada Ltd.

Abstract

This paper describes optimization techniques recently applied to the Just-In-Time compilers that are part of the IBM® Developer Kit for Java™ and the J9 Java virtual machine specification. It focusses primarily on those optimizations that improved server and middleware performance. Large server and middleware applications written in the Java programming language present a variety of performance challenges to virtual machines (VMs) and just-in-time (JIT) compilers; we must address not only steady-state performance but also start-up time. In this paper, we describe 12 optimizations that have been implemented in IBM products because they improve the performance and scalability of these types of applications. These optimizations reduce, for example, the overhead of synchronization, object allocation, and some commonly used Java class library calls. We also describe techniques to address server start-up time, such as recompilation strategies. The experimental results show that the optimizations we discuss in this paper improve the performance of applications such as SPECjbb2000 and SPECjAppServer2002 by as much as 10-15%.

1. Introduction

The steady growth in the size and complexity of large server and middleware applications written in the Java™ programming language[8] offers continuing performance challenges to virtual machines (VMs) and just-in-time (JIT) compilers. One source for these challenges is that program maintenance has become at least as important as performance for this class of applications. This stronger emphasis on program maintenance has encouraged programmers to make more widespread use of the many features available in the Java programming language that support software engineering efforts, such as: multiple inheritance via interface classes, polymorphic virtual method invocations, and complex exception handling mechanisms such as `finally` blocks. While these features are convenient to use from a Java programmer's perspective, they can impose a significant runtime performance overhead and thus they challenge JITs and VMs to provide features that mitigate that overhead. Moreover, addressing these software engineering related overheads is doubly difficult since many large middleware applications do not have obvious hotspots on which to focus compilation or performance analysis resources.

Even beyond this particular challenge, however, server and middleware applications have many other features that negatively impact their performance. In this paper, we describe 12 separate optimizations and features that have been implemented in IBM products to accelerate the performance of these applications. Examples of these features include: optimizing synchronization to reduce repetitive locking and unlocking on the same object to improve scalability, and optimizing Java class libraries to improve the performance of, for example, computing transaction time-stamps.

Steady-state performance, however, is not the only important performance factor. Application servers, for example, must be able to start quickly when a machine is rebooted to avoid costly interruption of service. Fast start-up time also greatly enhances productivity for application developers who regularly must restart the application server as part of the usual edit-compile-debug development cycle. We show in this paper how the JIT compiler can employ different recompilation strategies to significantly improve server start-up time.

This paper describes recently implemented optimization techniques used by the Just-In-Time compilers that are part of the IBM® Developer Kit for Java and the J9 Java virtual machine specification, focussing primarily on those optimizations that improved server and middleware performance. We report performance results on three platforms for industrial-strength implementations of these features rigorous enough to stand up to our customers' applications.

The rest of the paper is organized as follows. In Section 2, we discuss three Java coding conventions we have observed in customer applications and detail some of the performance overheads incurred in each case. In Section 3, we describe eight optimizations that specifically target server performance. Similarly, in Section 4, we describe four optimizations we have found to be important for middleware applications. We present the performance results achieved by these techniques in Section 5 using a well-known server application (SPECjbb2000), a middleware application (SPECjAppServer2002), as well as an XML parser and a cryptographic benchmark program. Finally, in Section 6, we summarize the paper's contributions.

2. Java Coding Observations

Through the process of addressing customer defects, our teams are exposed to large amounts of Java code written by our customers. We present, in this section, what we believe are three important observations about the code being employed that have a direct impact on the performance of Java applications: 1) a trend towards bytecode generation rather than Java code translated by `javac`, 2) the more frequent use of `finally` blocks, and 3) the common use of exceptions.

2.1. Bytecode Generation

Bytecode generators other than `javac` are becoming more prevalent. These range from relatively simple tools such as JavaServer Pages (JSP) servlet generators to full-fledged compilers such as extensible stylesheet language transformation (XSLT) compilers. The bytecode streams generated by these tools may have performance implications throughout the virtual machine and in particular in their interactions with JITs. Some of these interactions can be ameliorated through advances in JIT implementations but some require care in the bytecode generator implementation.

One particular factor that should be taken into account is that very long compilations can cause undesirable system behavior. For example, if a very large method is invoked as part of servicing an RMI request, the time taken to compile that large method might trigger a time out elsewhere, the effect of which could cascade through the system. For this and other reasons, most JITs have built-in limits on various phases such as method inlining to prevent compile times from becoming prohibitively long. However, methods that are individually very large, such as some generated JSP servlets, can still induce long compilations, which can result in performance problems as outlined above.

A second important consideration is that automatically generated bytecodes tend to exploit the more powerful aspects of the Java programming language more fully than programs written directly in Java. Tools that compile domain-specific languages to bytecode quite naturally build their own abstractions to represent elements of interest in their language. For example, if language X supported subroutine calls, an X-to-bytecode compiler might use an explicit representation of a stack frame. It is often easier for such a compiler to use type-opaque representations of these abstractions, which are supported by the Java language, to simplify both the bytecode generation model and the supporting run time. JITs, on the other hand, are better able to optimize the use of concrete data types and abstract types typically used by Java programmers. Work may be required in both JITs and bytecode generators to ensure that generated bytecodes benefit from overall optimizations such as de-virtualization and method specialization.

2.2. Finally Blocks

`Finally` blocks provide a mechanism whereby a Java programmer can guarantee an action is performed regardless of how control leaves a method: whether via a normal return path or via an exception. One way that this mechanism is employed in middleware applications is to guarantee that a tracing function will be called if it is necessary to document that the method has completed executing. In this coding pattern, most of the code in the method is enclosed within a `try` region and the tracing code is placed in a `finally` block for that `try` region. This organization guarantees consistency of tracing information.

The control flows associated with the use of a `finally` block can be quite complex, complicating the construction of efficient and effective traversal orders for the JIT, which can necessitate more iterations than otherwise necessary to solve dataflow equations. The resulting increase in JIT compilation time can affect the program's performance. Even worse, if the control flows are sufficiently complex (consider nested `finally` blocks, for example, which can and do occur after aggressive inlining) the JIT may decide not to, or be unable to, apply certain optimizations that can have profound performance implications.

Although `finally` blocks are convenient from a programmer's point of view, they are less so for the JIT compiler. We recommend against using `finally` blocks in performance-critical codes unless there is a valid engineering reason to do so.

2.3. Exceptions

Despite the wealth of evidence detailing the hidden cost of processing exceptions, we still encounter applications where exceptions are thrown in commonly executed paths. While JITs expend considerable effort to improve the delay between the time an exception is thrown and the time that the handler for that exception begins to execute, the delay is still significant and several orders of magnitude larger than executing, for example, a branch. In particular, throwing an exception that will be caught higher in the stack can be particularly expensive because each frame must be searched both for a catcher and for locked objects that must be unlocked. The latter problem is typically addressed by having an artificial catch block that unlocks the object and then rethrows the exception. If the "real" catcher is many frames above the frame where the exception was originally thrown and there are many objects to unlock in the intermediate frames, the exception will be even more expensive to process; it will involve throwing the exception several times.

3. Server Performance Work

In this section, we describe eight areas in which we have made enhancements (either in the VM or in the JIT) to improve the performance of server applications: `newInstance`, `String.indexOf()`, `System.currentTimeMillis()`, `System.arraycopy()`, object allocation inlining, lock coarsening, thread-local heap batch clearing, and utilizing the Intel® SSE instructions. Many of these enhancements target, in particular, the SPECjbb2000 [16] benchmark.

3.1. newInstance

The `java.lang.Class.newInstance` method returns an instance of the same type as the `java.lang.Class` object passed as the parameter to `newInstance`. The `newInstance` method in the class library calls a native `newInstanceImpl` method that does three things: first, it checks for the presence of a default constructor for the class being instantiated; second, it checks that this default constructor is accessible to the caller; and third, it invokes the constructor. Several factors make this process inefficient:

1. The expensive invocation to the native `newInstanceImpl` method.
2. The search for the default constructor.
3. Verifying that the constructor is visible in the caller's context may involve walking the stack to determine that the constructor is not currently being executed.
4. The expensive callback into the Java constructor from the native method.

Because `newInstance` is commonly used, it warrants a special slot in the virtual function table (VFT) of each class. This entry represents a class-specific `newInstanceImplThunk,` which can perform the allocation, avoiding both the native invocation (#1) and the callback to the Java constructor (#4). We also avoid the accessibility check (#2) altogether for classes with public default constructors. Furthermore, we can inline the constructor into the caller and avoid even overhead of a JITed code invocation. Even better, the JIT can use value profiling information[5] to inline the `newInstanceImplThunk` invocation with appropriate runtime guards, which has the benefit of exposing the allocation to the JIT's optimizer. Once exposed, optimizations such as escape analysis[6] can transform the heap allocation into a stack allocation if the created object does not escape to another thread or to the method's caller.

3.2. String.indexOf()

Numerous algorithms, such as Boyer and Moore's technique[4], exist to perform the semantics specified by the `String.indexOf()` method in the Java class library. Many of these algorithms operate in two phases: first they compute some meta-data based on the target string and then they use this to rapidly iterate through the source string. Our analysis of benchmarks such as SPECjbb2000 revealed that `indexOf` is frequently invoked with short constant pattern strings. This situation is ideal for the application of Boyer and Moore's algorithm because the compiler can statically compute the meta-data associated with the pattern string, leaving only the rapid phase to be executed in response to the actual invocation of `indexOf`.

3.3. System.currentTimeMillis()

The `java.lang.System.currentTimeMillis()` method is often called by transaction-based server applications such as SPECjbb2000 that repeatedly build time stamps. This method is expensive because:

1. There is an expensive call from JITted code to a native method.
2. The method returns a long value that must be held in a register (increasing register pressure) or returned via the stack (which may be slower to access).

We optimize these costs by generating a platform-specific inline code sequence to replace calls to `System.currentTimeMillis()`. The inline sequence sets up the arguments according to the system linkage convention and then directly calls the appropriate OS timing routine: `GetSystemTimeAsFileTime()` in a Microsoft® Windows® environment, for example, or `gettimeofday()` in a Linux environment. The inlined sequence stores the result directly as required by the application's use of `currentTimeMillis()`.

3.4. System.arraycopy()

One of the most frequently used intrinsics in Java middleware applications is `System.arraycopy()`. The optimal code sequence for this method is platform-specific and varies with the actual size of the array to be copied.

To avoid the frequently suboptimal generic arraycopy implementation for specific calls to `System.arraycopy()`, the compiler can either generate code inline or invoke tuned versions of `System.arraycopy()` provided in the compiler's runtime. The compiler can, of course, implement both approaches and evaluate the cost trade-off at code generation between the call overhead to the tuned library code versus the code expansion cost of the inline code to copy the array.

If the size of the array to be copied is constant, the JIT can easily generate the best possible instruction sequence. When the size isn't known, however, it has to either generate an instruction sequence that will perform best on average (like the generic implementation of the arraycopy intrinsic) or it can employ value profiling to determine the most frequent array size. In a subsequent compilation for the method, the JIT can then special-case that most frequent size with an inlined code sequence and rely on a more generic code sequence for other sizes. Since most of the arrays copied in middleware applications are shorter than 256 bytes, the JIT can also choose either to use a general instruction sequence that works best for shorter array sizes, or to use a runtime test to exploit the benefit of the faster arraycopy instruction sequence for cases where the faster sequence will compensate for the test overhead.

When copying arrays of references, there are additional optimizations that are performed. These optimizations fall into two broad categories deriving from: 1) the garbage collector design, and 2) the need to enforce the Java rules [11][12] for reference assignment compatibility.

To maintain correctness, a generational garbage collector[18] must detect when a reference to a nursery (or "new space") object is stored into a tenured (or "old space") array object. For certain array copies, the JIT can prove that checking for such stores is unnecessary. The simplest optimization opportunity of this class, for example, is copying regions within a single array. Clearly the array cannot be both in the nursery and tenured at the same time. A second opportunity in this class occurs once one nursery object reference has been stored into a tenured object/array. Once the tenured array has been added to the list of objects that must be scanned when garbage collecting the nursery, the remaining elements of the array are copied without checking for nursery objects[1].

The second class of optimizations results from the JIT's existing type propagation capabilities. It is often the case that the JIT is able to prove that there will be no exceptions raised by storing the elements of one array into another.

There are two effects of these optimizations that improve performance. First, proving that it is unnecessary to load either the class object or the garbage collection bits from the header of the object being copied reduces memory traffic into the cache during the copy, which in turn reduces the cache pollution effects of the copy. Second, by eliminating the control flow from the copy loop, more efficient copy mechanisms supported by most CPU architectures are enabled.

3.5. Object Allocation Inlining

Virtually all versions of the IBM Developer Kits have included facilities that enable JITs to perform common object allocations using a thread local heap (TLH). A TLH is a small region of the heap that is temporarily assigned for the exclusive use of a particular thread. When no further allocations are possible in a TLH, the region resumes its status as a regular part of the heap.

[1] The check can be omitted so long as collection cannot occur during the array copy.

Since each thread has its own TLH, no synchronization is needed to allocate the majority of objects. When processing modestly sized object allocation requests, the JIT will generate fast-path code to allocate the objects in the TLH. The initial implementations of this strategy targeted the case where the allocation is satisfied from the TLH as the fast-path. The generated code was quite effective in this case, but each allocation site had associated with it a series of return code checks and recovery sequences to handle the less-frequent cases where the allocation could not be satisfied by the TLH.

Analysis of many benchmarks revealed that the execution time spent in the return-code checking code could be substantial. The Developer Kit was enhanced so that, when the fast-path code fails to allocate storage from the TLH, it invokes a JIT service routine. This routine, which is part of the garbage collection (GC) component, will always return either a new object or an `OutOfMemoryError` indication. The latter can be checked quickly by the fast-path code. The service routine handles the necessary bookkeeping, including possibly collecting unused objects, as well as memory synchronizations. As a result, the fast-path code sequence is shorter, which results in better instruction cache utilization. Furthermore, in most cases where the original TLH storage allocation request fails, fewer compares and branches are executed.

3.6. Lock Coarsening

Because the JIT aggressively inlines invocations when compiling a method, it is not uncommon to synthesize a block of code containing several lock and unlock operations applied serially (i.e., not in nested fashion) to the same object. These lock and unlock operations result from inlining synchronized methods into the method being compiled. When we analyzed SPECjbb2000, we noted a method that executed 6 or 7 repetitive lock and unlock operations on the same object, depending on the path executed.

Synchronization has two negative impacts on program performance. First, the lock and unlock operations themselves are expensive, in most cases requiring expensive atomic memory accesses. Second, locks act as barriers to optimizations in the JIT, to processor instruction schedulers, and to the reordering ability of most modern processors. To alleviate these two problems, we implemented a lock coarsening optimization that eliminates many of the intermediate unlock and lock operations, leaving only one lock to be executed early in the method and as many unlocks as there are paths exiting the locked region. This optimization must be performed with great care, however, because holding a lock for a substantially longer time can increase contention and therefore reduce an application's scalability. Even worse, without proper care, deadlock opportunities can be created that make the application hang.

We implemented a computationally efficient lock coarsening pass that does not unduly increase contention or create deadlock opportunities. The pass only acts on synchronization resulting from inlining synchronized methods and so it will not interfere with hand-crafted synchronization implemented by a programmer via a synchronized block. Although there has been previous work in this area [1][3][7], our implementation is novel because of its aggressiveness and its computational efficiency. Unfortunately, space restrictions prevent us from providing a detailed description of the technique in this paper.

3.7. Thread-local Heap Batch Clearing

The IBM JITs employ a thread-local heap (see Section 3.5) to reduce contention on the heap lock. On some processors, it can be more efficient to initialize the entire contents of the thread-local heap to the value zero when it is allocated rather than initialize those values for each individual object allocation. In particular, the IBM PowerPC® architecture allows an entire cache line at a time to be initialized to zero. The IBM JITs use this batch clearing approach only for processors that have efficient architectural support for initializing large blocks of memory. For other processors, the objects allocated from the local heap are initialized individually.

3.8. Intel® SSE Instructions

When Intel introduced the Pentium® III processor, it included Streaming SIMD Extensions (SSE), a technology designed to accelerate applications such as 2D/3D graphics, image processing, and speech recognition. It includes a file of eight 128-bit XMM registers. The SSE architecture specifies that each XMM register holds four single-precision floating-point values. A further extension (SSE2) introduced in the Xeon™ and Pentium 4 processors, allows each register to hold a pair of double-precision floating-point values (or four single-precision values). The extensions include a rich set of instructions for inserting and extracting data from XMM registers and, of course, for manipulating the data therein.

Fully exploiting the SIMD capabilities of these extensions in a Java environment is nontrivial. The difficulty arises primarily because the extensions perform best when the data being moved in and out of XMM registers have specific alignment characteristics and the instructions are used in a streaming fashion. Keeping salient data aligned in an environment with active garbage collection (and object relocation) is a significant challenge. Furthermore, relatively few applications are streaming in nature. Without surmounting this challenge, however, the SSE architecture can still be used to good advantage. If each XMM register is treated as a single value (single- or double-precision), the extensions form an excellent scalar floating-point computation engine. Furthermore, it is easier for the JIT to manage a set of eight orthogonal registers than the x87 FPU stack, especially in the presence of registers whose live ranges span control flow boundaries. Our JIT compiler therefore favors the SSE extensions over the traditional floating point mechanisms when targeting X86 CPUs. An exception to this rule is in cases where SSE2 is not available and a particular method requires frequent conversions between single and double precision.

4. Middleware Performance Work

We describe in this section four items on which we have recently focussed to improve the performance of middleware applications like SPECjAppServer2002[15]: reducing application server start-up time via recompilation strategies, improving interface invocations via polymorphic inline caches, recognizing when 64-bit long variables are used to (inefficiently) perform unsigned 32-bit computations, and code reordering to reduce branch mispredictions and instruction cache misses.

4.1. Application Server Start-up Time

The start-up time of the application server is critical for many reasons, including quick recovery from server failures and shorter development time since the server is often started at least once per edit-compile-debug cycle.

This requirement is a challenging aspect of the quality of the JIT compiler because the speed of the compiler itself is critical to good performance, rather than the quality of the generated code. There are two conflicting factors that have significant effect on the start-up performance. First, the compiler itself should take as little time as possible. Second, we want to generate fast code to reduce the application execution time. These two factors must be carefully balanced and tuned to minimize start-up time.

We approached this problem by applying various recompilation strategies with different optimization levels and correspondingly different compile times. However, application server start-up frequently involves transient behavior: some methods are hot for a short period of time, which confuses the profiler and causes the JIT to recompile the method at an optimization level higher than is truly justified. The JIT expends considerable resources compiling such a method, but the method does not execute frequently after it is recompiled and so the compilation effort is not rewarded. To reduce incidences of this problem, the JIT compiler has additional heuristics to estimate the expected gain to be achieved by recompiling methods at higher optimization levels. These heuristics dampen the aggressiveness of compilation optimization early in the process, which benefits application server start-up time.

4.2. Polymorphic Inline Caches

Java invocations can be polymorphic: when a method is invoked, the actual target is determined by the runtime type of the receive object. This feature benefits programmers by allowing the design of clear, reusable and maintainable code. But this benefit comes with the additional and sometimes not insignificant cost that each such polymorphic invocation requires the system to decode the type of the receiver object before the appropriate target can be invoked. Determining the correct target for an invocation via an interface method can be particularly expensive because the Java language allows a class to implement multiple interfaces. Large middleware applications often use interface-based polymorphism so that they can be easily extended with additional features, and therefore these applications can suffer from the performance impairment because of the polymorphic call overhead

An efficient technique to reduce the effect of the interface invocation overhead is a polymorphic inline cache (PIC) [2][9][10][17] which is a dynamic code or data structure that can cache a particular target call site and perform quick subsequent invocation at the same call site, without invoking the target lookup routine. The compiler generates and maintains a small cache containing usually two to four entries of the most frequent actual targets for the polymorphic invocation. By employing such a cache for each interface invocation site, the majority of the invocations can be quickly dispatched. Since most polymorphic invocation sites have only a few targets[2], caching a small number of targets for low-overhead invocation can greatly mitigate the total overhead of the full polymorphic call sequence.

The polymorphic cache is initially empty after the method is compiled. As the program continues to execute, the targets invoked from the site are recorded with the type of the invocation's receiver object into the cache. Subsequent invocations with that same receiver object type will be directly dispatched to the cached target. If the cache is full and the receiver object's type does not match any of the cache entries, a full lookup and dispatch is performed (the "slow" path). Profiling can be used to order or even evict entries in the cache according to frequency. The profile data can also be used to recompile the method containing the call site if inlining the most frequently executed target would be beneficial.

4.3. Unsigned Arithmetic for Cryptography Applications

With the introduction of SSL (Secure Sockets Layer) and its implementation in middleware application servers written in Java code, the performance of transactions that use the security protocol becomes as important as traditionally non-secure transaction operations. This secure layer overhead can significantly impair the performance of a secure application compared to a non-secure implementation.

The JIT can affect the performance of this secure layer because of a specific characteristic of the Java code often used to implement these cryptographic codes. A frequently appropriate data type to use in these codes is unsigned 32-bit integer, which is unfortunately not directly available in the Java programming language. To implement many cryptographic algorithms, therefore, Java programmers have resorted to using the 64-bit signed long data type to hold 32-bit unsigned data, which on 32-bit architectures can cause a significant performance slowdown.

To address this problem we have added support in the JIT compiler to recognize long arithmetic operations that are used to implement 32-bit unsigned arithmetic. If the JIT can prove that the upper 32 bits of a 64-bit computation are zero, then more efficient instruction sequences can be used on some platforms to accomplish that computation than the naïvely generated 64-bit code.

4.4. Code Reordering

As a debugging aid, the methods of many middleware applications include tracing code that conditionally executes at the beginning and/or end of the method to document how the application is executing. Since the tracing information is typically voluminous, this code is typically executed when debugging the application and is only rarely executed in a production environment. An example of this style of coding is as follows:

```
void aMethod(...) {
   if (_trace.entryTracing()) {
      _trace.entry("aMethod");
      // log arg info
   }

   // do the work of aMethod

   if (_trace.exitTracing()) {
      _trace.exit("aMethod");
       // log effect
       // log return value info
   }
}
```

While these conditional code snippets provide useful debugging information when there is a problem, they can also impose a performance penalty in a middleware application even when they do not execute. There are three aspects to this penalty:

1. The additional code can perturb the JIT's allocation of machine resources such as registers.
2. The instruction fetch engine for many modern processors will (incorrectly) predict by default that the forward branch around the code snippet will not be taken.
3. The effective user code is inefficiently and unnecessarily spread over additional cache lines and memory pages.

Fortunately, the first problem is easily overcome because the JIT usually has access to profile information indicating which paths are not frequently executed. In many cases, this information enables the JIT to avoid dedicating resources to those parts of the code that are rarely executed. In the above example, the JIT would not likely inline the `entry()` and `exit()` invocations if those paths are marked as rare code.

In contrast to the first problem, the second problem is significant. The condition test is turned into a forward branch around the code outputting the trace information. Most current processors, by default, assume that forward branches are not taken. To alleviate this problem, current processors also include prediction tables that remember what happened the last time a branch was executed. These tables are used by the processor to predict future outcomes for branches based on the information about earlier branch outcomes stored in the table. Unfortunately, these tables are a scarce resource, and middleware applications can be so large that by the time a branch instruction is fetched a second or subsequent time, the history for that instruction has already been ejected. When there is no historical data in the table for a branch, the processor falls back to the default not-taken prediction for a forward branch. The processor then speculatively executes the tracing code only to discover several cycles later that it had mispredicted the branch and must throw away all the work it has done. Recovering from a branch misprediction can take more than 10 processor cycles, depending on the processor, plus any work it has done since the prediction was made must also be discarded.

The third problem results in instruction cache and, potentially, TLB misses because the code executing is typically spread over more cache lines or memory pages than is necessary.

To solve these three problems, the JIT reorders the basic blocks of each method to make each block's most commonly executed successor its fall-through successor. Since the JIT has information about which direction each branch favors, branches with a relatively biased outcome such as debugging tests can be flipped so that the processor's default not-taken prediction for forward branches becomes the correct prediction. Some processors do not even try to remember a branch if its prediction matches the default prediction, which means this optimization can have a secondary performance benefit because more branches can be maintained in the prediction tables than before. Finally, because the commonly executed code appears close together in memory, fewer instruction cache and TLB misses are expected. Although reordering code along hot paths is not a new idea [13], we have found it to be particularly effective in the context of a JIT compiler.

5. Results

In this section, we report the benefits we have seen in a group of benchmarks relevant to server and middleware application performance. The improvements we report have not been collected at a single point in time but rather over the course of the recent product development cycle for the practical reason that it is not always possible or reasonable to maintain code to selectively enable or disable particular features. In a production JIT compiler, wherever serviceability is not improved by such code, it is eliminated after a testing cycle to simplify code maintenance.

The impact of this practise on the results of this section is that individual improvements should not be considered additive. If two improvements are measured at 3% and 5%, one should not conclude that the two improvements together would provide an aggregate 8% speedup; the two measurements have implicitly different baselines. Furthermore, IBM maintains more than one Java JIT product and some improvements are specific to one particular technology base. The two products we have evaluated are the IBM Developer Kit for Java and the J9 Java virtual machine.

Moreover, some of the improvements described in this section are platform-specific. These results should not be carried across to other platforms as the improvements will vary considerably from platform to platform; the work done for one platform may be irrelevant or even harmful for another. To the best of our knowledge, we have not excluded any result available to us simply because it degraded performance.

In all cases, we have made an effort to collect together improvements that can reasonably be considered together by platform or by underlying JIT technology, though these improvements should never be interpreted in any additive sense except where explicitly noted.

The results in this section are organized by benchmark. The improvements that had an impact on each benchmark are listed in each subsection. The benchmarks we discuss are: SPECjbb2000[16], SPECjAppServer2002[15], XML Parser[14], a cryptography-based micro-benchmark, and IBM WebSphere® Application Server start-up.

5.1. SPECjbb2000

The SPECjbb2000 benchmark models a pure server application without using an application server or database tier. The benchmark encodes the logic and database for a business managing orders, inventory, deliveries, and payments. It is inspired by, but not directly comparable to, the TPC-C database workload. The benchmark progressively increases its workload to test the throughput supported by a particular system by increasing the number of operating warehouses over time, starting at 1 warehouse and ending at 2N warehouses where N is usually the number of processors in the machine. The results reported below are improvements to the final score, which is an aggregate metric that primarily tracks the average of the throughputs achieved at warehouses P through 2P, where P is the warehouse at which the peak throughput was measured (usually P = N).

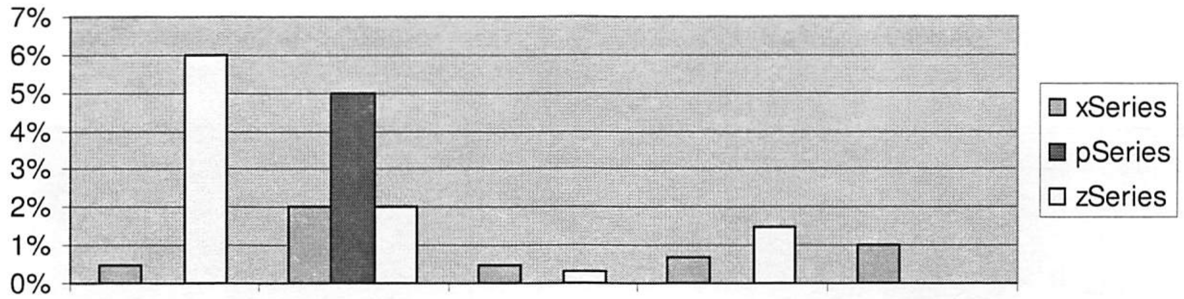


Figure 1: SPECjbb2000 Improvements, IBM Developer Kit for Java

Many of the features we've described in Section 3 improve the performance of the SPECjbb2000 benchmark. The benefit for each such feature for three platforms (xSeries®, pSeries®, and zSeries®) and for the IBM Developer Kit for Java product is shown in Figure 1. The percentage improvement is in the self-reporting score relative to the same score without the specific feature. Note that some bars are not present in this graph because those features have not been implemented for that particular platform or for this particular product. The xSeries results were measured on a 4x2.8Ghz Pentium 4 system with Intel Hyper-threading enabled running Microsoft Windows 2000. The pSeries results were measured on a 16-way 1.1Ghz POWER4 p670 system running AIX® 5.1. The zSeries results were measured on a variety of different machine configurations and so these results in particular should not be compared directly against one another.

For the J9 virtual machine product, we report the improvements found in Figure 2. Again, some bars are not present because that feature was not implemented for a particular platform or in this product, or because that feature had no benefit on that particular platform.

5.2. SPECjAppServer2002

The SPECjAppServer2002 benchmark is a large-scale middleware application benchmark that models the complete business processes of a Fortune-500 company, from customer ordering and invoicing through inventory and supply chain management to manufacturing. The benchmark exercises both an application server and a database tier, which can reside on the same machine, on different machines, or each tier can be distributed across many machines. It is a strenuous test of Enterprise JavaBeans (EJB) 2.0. From the JIT's perspective, our main ability to improve this benchmark is within the application server, where a very flat and widely distributed profile greatly hinders the identification of "bang-for-the-buck" performance improvements.

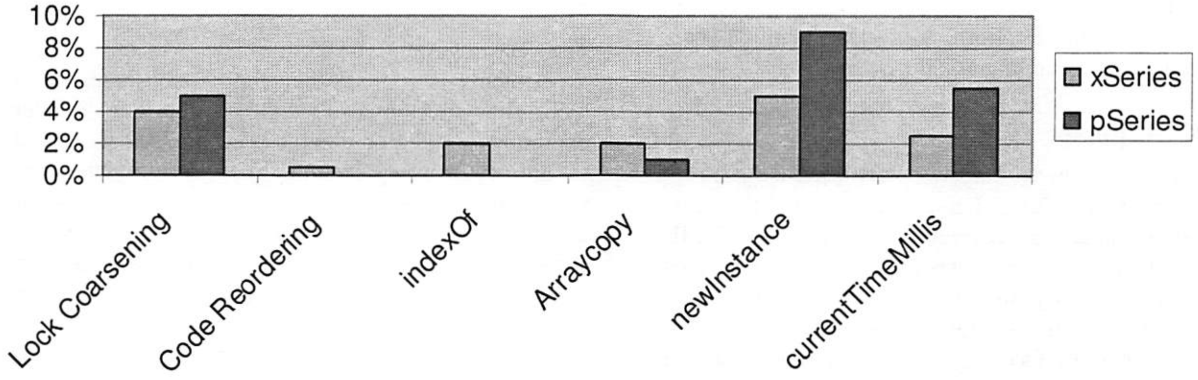


Figure 2: SPECjbb2000 Improvements, J9 virtual machine

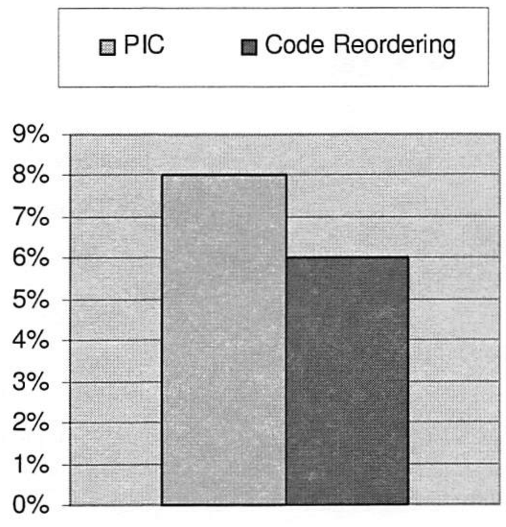


Figure 3: SPECjAppServer2002 Improvements

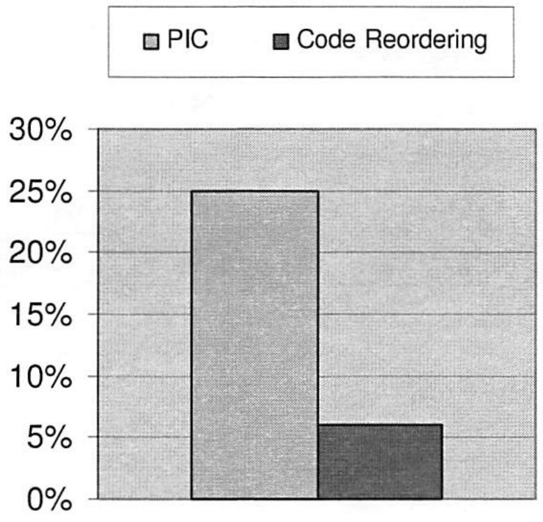


Figure 4: XML Parser Benchmark Improvements

Our investigation of the SPECjAppServer2002 benchmark is still in its preliminary stages. Nevertheless, we have observed that two of the improvements discussed in Section 4 have significantly improved the performance of this benchmark: polymorphic inline caches and code reordering (see Figure 3).

In this particular case, these two measurements can be combined since the improvement we have seen by enabling both of these features in the IBM Developer Kit for Java was 14%. In the context of the incredibly flat profile for this application, both results are even more impressive. We are continuing to investigate improving the performance of this benchmark. As for the SPECjbb2000 benchmark, the percentage improvement is in the self-reporting score relative to the same score without the specific feature. The performance improvements were measured on an xSeries 4x2.8Ghz Pentium 4 system with Hyper-threading enabled running Windows 2000.

5.3. XML Parser

The XML Parser benchmark we have used is based on the Apache Xerces XML Parser for Java [14] that tests parser performance on different combinations of XML data sets. One of the main design goals for the Xerces parser is extensibility; the implementation of the parser relies heavily on abstract classes and method invocations via interface classes. An example of such a class is `AbstractSAXParser`. Given this design approach, proper implementation of Polymorphic Inline Caches (PICs) can yield significant improvements as measured on an xSeries 1.6 Ghz Pentium 4 uniprocessor running Microsoft Windows 2000, which are presented in Figure 4, measured on the IBM Developer Kit for Java product. Note that the scale of the y-axis is different than that of the adjacent Figure 3. We tested the XML Parser benchmark using several different sample XML files with sizes ranging from 1 KB to 5 MB. The benchmark parses each of the files a number of times and calculates a ratio score reflecting the number of elements parsed per second for each data set. The benchmark also computes an average score by combining and interleaving the parsing for each sample XML file into a single run. Figure 4 also includes the improvement in the parser throughput because of the code reordering technique described in Section 4.4.

5.4. Cryptography

The cryptography benchmark we used for our experiments is a micro-benchmark consisting of a loop that performs extended precision integer operations. The loop computes A*B+C where A and C are 1024-bit precision integers simulated by vectors of 32 unsigned 32-bit values using arrays of the Java long data type[2] and B is a 32-bit unsigned number. The loop body contains one long addition and one long multiplication operation. On a 32-bit

[2] Java has no unsigned integer types, so longs are typically used to hold unsigned 32-bit values.

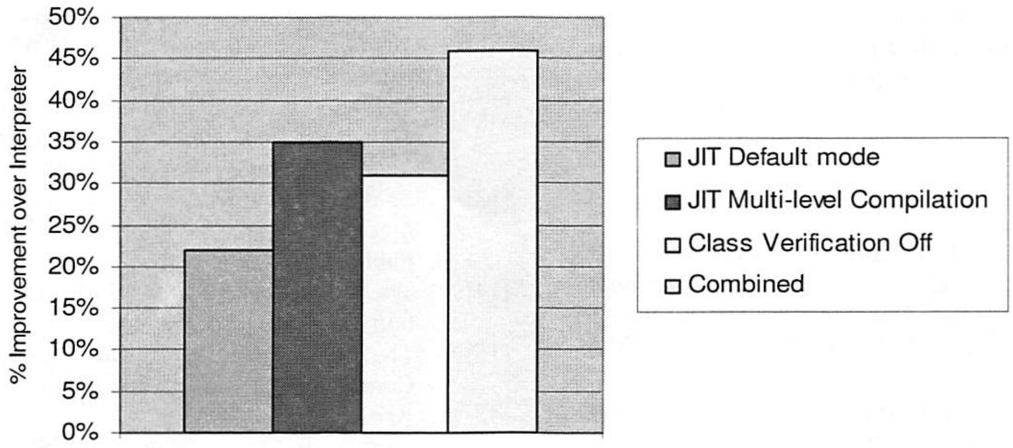


Figure 5: Application Server Start-up Improvements

platform, the long add operation naïvely expands to two 32-bit operations and four 32-bit loads. Similarly, the long multiplication, depending on the instruction set architecture (ISA) of the platform, naïvely requires three 32-bit multiplies (two 32-bit by 32-bit producing low order 32-bit results and one full 32-bit by 32-bit multiply with 64-bit result), two adds to compute the high order 32-bits of the product from the three multiplies, and four 32-bit loads.

The JIT compiler is able to discover that the long operands are actually 32-bit unsigned quantities and therefore that the high order 32-bits of each long used to hold an unsigned 32-bit value are provably zero. This discovery enables the JIT to dramatically simplify the computation and memory traffic that occurs in computing this inner product. After the JIT recognizes the 32-bit unsigned multiplication, the computation is performed by one 32-bit by 32-bit multiply with 64-bit product, no adds, and two loads. Similarly, the 64-bit add operation in the loop is reduced from two adds and four loads to two adds and two loads.

Transforming these computations as described above results in an almost 50% performance improvement, which, in the case of X86, is due to the reduced computation and operand driven memory traffic as well as the complete elimination of spill and fill instructions in the loop. The results are based on measurements performed on an xSeries 1.6 GHz Pentium 4 uniprocessor running Microsoft Windows 2000.

5.5. Application Server Start-up

As the basis for application server start-up performance analysis we chose the IBM WebSphere Application Server product, where we report the levels of improvement that can be achieved by applying multi-level optimization recompilation strategies. Since significant class loading occurs at server start-up, we show that significant additional performance improvement can be achieved by reducing the class loading cost by turning off class verification. Figure 5 shows the percentage improvements relative to the time taken by the interpreter to boot the application server to its ready state. The results were collected by running the IBM WebSphere Application Server product with the J9 virtual machine on an xSeries 1.6 GHz Pentium 4 uniprocessor on Microsoft Windows 2000.

6. Summary

In this paper, we have made three basic contributions. First, we made three observations regarding Java coding practices among our customers that directly impact the performance capabilities of JITs and VMs: bytecode generation, the more prevalent use of finally blocks and the continuing frequent use of exceptions. Second, we described 12 different optimizations and features that our teams have developed recently to improve the performance of both server and middleware applications. Third, we presented results that show the benefits of robust implementations of these optimizations for a variety of applications. These results demonstrate both the effectiveness of the features we have implemented as well as the level of improvements that can reasonably be expected for robust implementations in a high performance production JIT.

Acknowledgements

The authors would like to thank the members of the IBM Java JIT and VM development teams whose work is described in this paper. In particular, we would like to acknowledge the feedback and advice from Alan Adamson, Mike Fulton, and Derek Inglis in the preparation of this paper. We also thank the reviewers for their time and effort to evaluate this paper.

References

[1] J. Aldrich, E. G. Sirer, C. Chambers, and S. Eggars. *Comprehensive Synchronization Elimination for Java.* In Science of Computer Programming, Volumn 47, Issue 2-3, May 2003.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and Derek Lieber. *Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless.* In ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Oct 2001.

[3] J. Bogda and U. Holzle. *Removing Unnecessary Synchronization in Java.* In Conference on Object-Oriented Programming, Systems, Languages, and Applications, Nov 1999.

[4] R. Boyer and S. Moore, *A Fast String Searching Algorithm*, CACM 20(10), pg 762-772 (1977).

[5] B. Calder, P. Feller, and A. Eustace. *Value Profiling and Optimization.* Journal of Instruction Level Parallelism, Mar. 1999.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. *Escape Analysis for Java.* In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. Denver, Colorado. Nov. 1999.

[7] P. Diniz and M. Rinard. *Lock Coarsening: Eliminating lock overhead in automatically parallelized object-based programs.* In Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing. San Jose, CA, Aug 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[9] D. Grove, J. Dean, C. Garrett, and C. Chambers. *Profile-guided receiver class prediction.* In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 108-123, Oct 1995.

[10] U. Holzle, C. Chambers, and D. Ungar. *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches.* In P. America, editor, Proceedings ECOOP'91, LNCS 512, pages 21-38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1996.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition.* The Java Series. Addison-Wesley, 1999.

[13] S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers. 1997.

[14] The Apache XML Project, 2001. http://xml.apache.org/xerces-j

[15] The Standard Performance Evaluation Corporation. SPECjAppServer 2002 Benchmark. http://www.spec.org/jAppServer2002, 2002.

[16] The Standard Performance Evaluation Corporation. SPEC JBB 2000 Benchmark. http://www.spec.org/osg/jbb2000, 2000.

[17] J. Vitek, and R. N. Horspool. *Compact dispatch tables for dynamically typed object oriented languages.* In Proceedings of International Conference on Compiler Construction (CC'96) pages 281-293, Apr. 1996. Published as LNCS, vol 1060.

[18] P. Wilson. *Uniprocessor Garbage Collection Techniques.* In Proceedings of International Workshop on Memory Management. Springer-Verlag. Saint-Malo, France. 1992.

Trademarks

Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing

Ali Raza Butt, Xing Fang, Y. Charlie Hu, and Samuel Midkiff
Purdue University
West Lafayette, IN 47907
{butta, xfang, ychu, smidkiff}@purdue.edu

Abstract

The increased popularity of grid systems and cycle sharing across organizations leads to the need for scalable systems that provide facilities to locate resources, to be fair in the use of those resources, and to allow untrusted applications to be safely executed using those resources. This paper describes a prototype of such a system, where a peer-to-peer (p2p) network is used to locate and allocate resources; a Java Virtual Machine is used to allow applications to be safely hosted, and for their progress to be monitored by the submitter; and a novel distributed credit system supports accountability among providers and consumers of resources to use the system fairly. We provide experimental data showing that cheaters are quickly identified and purged from the system, and that the overhead of monitoring jobs is effectively zero.

1 Introduction

This paper describes a prototype of a complete system that allows the sharing of cycles across network connected machines. For any such system to be successful, certain core functionality must be provided to both applications and hosts: (i) the ability for an application to discover a machine (or cluster of machines) capable of hosting it (resource management and discovery); (ii) the ability for an application to run on a wide variety of machines without change (portability); (iii) the ability of a host machine to accept an application from an untrusted source, and execute it without being damaged (safety); and (iv) a mechanism for maintaining information about resources provided and consumed, and for ensuring fairness in the use of resources (accountability). These four functions must be accomplished in a distributed, scalable manner to enable large networks of machines and resources.

Resources that are available but that are not easily discovered are useless. *Peer-to-peer* (p2p) networks have achieved widespread use as a content discovery mechanism. We propose using these same mechanisms for resource discovery and job assignment for our cycle sharing framework. Moreover, because p2p networks are self-organizing, it is easy for nodes to join, and leave, without the necessity of a central administrative organization and human intervention.

The use of Java is extremely convenient, if not essential, for the second and third of these functions (portability and safety) and it makes the accounting significantly easier. Because overspecifying a host machine's characteristics reduces the number of viable execution targets for an application, the portability across execution environments provided by Java is essential. Moreover, Java's built-in sandboxing technology, and rich security infrastructure, allow applications of varying degrees of trust to be hosted without each host providing additional security mechanisms. Both of these attributes significantly lower the cost and risk for producers and consumers of cycles to join a network of shared resources. And research [25, 26] shows that there are no inherent reasons for not using Java for high performance computing.

Finally, a community of pooled resources will survive only as long as members are treated with a high (but not necessarily perfect) degree of fairness. In the physical world, money is used as a conveyor of information about one's contribution to the economy. Credit reports allow providers of services to judge the likelihood that they will be paid for those services and to hold consumers accountable for their debts. Incremental payment schemes are used in many large activities to bound the amount of risk for both providers and consumers of resources to the size of the incremental payment. In this paper we outline a low-overhead Java based mechanism to allow users to monitor the progress of their applications, and to determine if they are comfortable making partial payments for the progress of a job. Our credit mechanism provides a distributed, scalable system for making and accepting these (possibly partial and incremental) payments (credits, in the language of this paper.) Moreover, our system supports two other important functions: we allow both users and resource providers to determine, using their own criteria, the credit worthiness of others; and we allow credits held by one entity (user) to be traded to another, and used by that other entity to acquire resources or reduce its volume of outstanding credit. Just as the larger economy can function well with a certain amount of fraud and noise in transactions and accounting, so should economies involved in sharing of computational resources. Thus our goal is not to produce perfectly secure system, but instead sufficiently good systems to enable wide scale sharing of computational resources.

This paper makes the following contributions:

- A novel method for monitoring the progress of a Java application with low overhead that leverages important features of Java Virtual Machines (JVM);
- A novel method for issuing credits that is scalable and checkable by all participating nodes in a system;
- A system that allows the sharing of computational resources that builds on the Java properties of portability and safety, the credit system described in this paper, and scalable p2p networks;
- Experimental data showing the practicality of these techniques.

The rest of paper is organized as follows. Section 2 presents our proposed scheme for enforcing fairness in p2p cycle sharing systems. Section 3 discusses the details of our prototype implementation, and Section 4 measures the overhead and the effectiveness of our proposed scheme. Finally, Section 5 presents some related work, and Section 6 provides concluding remarks.

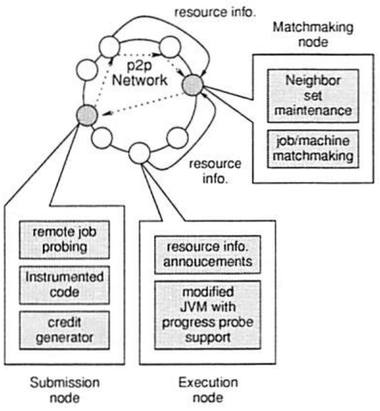


Figure 1: Overall design of the proposed scheme. Each node can be a submission node, or an execution node for submitted jobs.

2 System Design

This section first gives an overview of the system design, and then describes in detail how each of the building blocks implements the properties necessary for a usable cycle sharing system.

2.1 Overview

In order to obtain a cycle sharing system that prevents disproportionate consumption of resources, the system has to simultaneously provide the consumer with assurances that jobs are making progress on remote machines, and the resource provider with assurances that resource usage will be compensated. Figure 1 shows the design of our system. It uses a p2p substrate for two purposes: to organize all the participating nodes, and to locate remote nodes with available compute cycles. In the following we assume a remote node with free cycles has already been selected for remote execution.

Every participating node can be a resource consumer (submission node) or a provider (execution node). A node can fill both roles simultaneously as well, e.g., its jobs can be running on some remote nodes while jobs from other remote nodes are running on it. As a consumer, a node implements the following functions: (i) it implements a probing system that allows it to query a *reporting module* (a job that listens for, and accumulates information sent by the beacons which monitors the progress of an application) for progress reports on remote jobs; (ii) it has access to a special compiler that processes the information sent by beacons and check it for validity. In our current implementation all information sent by beacons is considered valid, and no information is extracted, but our design allows for more sophisticated reports (see Section 2.4.1 for details); (iii) it implements an accounting system to record the fact that it has consumed cycles on other nodes, and to issue digitally signed credits to the hosting machine when necessary. These reports are stored in the p2p network, and can be viewed by all nodes in the network.

As a resource provider, the execution node supports a trusted JVM whose dynamic compiler inserts *beacons* that send information about the progress of the application to the reporting module, mentioned above. The reporting module issues progress reports using this information.

Remote cycle sharing works as follows. First, the submission node creates a job to be run on remote resources. The p2p network is queried for a possible host node, the credit information for the node is checked, and if acceptable the job is submitted to the node. The VM and its dynamic compiler on the host node insert instrumentation and beacons into the program, which begins execution. The submitting node then periodically queries the reporting module (which can run on any node that can communicate with the job on the host node and with the submitting node). If the submitting node finds the job to be making progress, it issues a credit to the execution node. Credits are not issued if the job is not making satisfactory progress. If the submission node tries to cheat and not issue a credit to the host node, the host node can evict the job. Therefore, self interest motivates the submitting node to issue credits, and the host node to run the program.

2.2 Scalable resource management through p2p networks

To enable fault-tolerant, load-balanced sharing of compute cycles among the participating nodes, we use a structured p2p overlay network to organize the participating nodes and locate available compute cycles on remote nodes for remote execution.

2.2.1 Distributed Hash Tables

We briefly review current p2p overlay networks which previously have been used for supporting data-centric applications. Structured p2p overlay networks such as CAN[30], Chord[35], Pastry[32], and Tapestry[40] effectively implement scalable and fault-tolerant *distributed hash tables* (DHTs). Each node in the network has a unique nodeId and each data item stored in the network has a unique key. The nodeIds and keys live in the same namespace, and each key is mapped to a unique node in the network. Thus DHTs allow data to be inserted without knowing where it will be stored and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored. In the following, we give a brief description of one concrete structured overlay network, Pastry, on which our prototype is built. Detailed information can be found in [32, 7].

Pastry provides efficient and fault-tolerant content-addressable routing in a self-organizing overlay network. Each node in the Pastry network has a unique nodeId. When presented with a message with a key, Pastry nodes efficiently route the message to the node whose nodeId is numerically closest to the key, among all currently live Pastry nodes. Both nodeIds and keys are chosen from a large Id space with random uniform probability. The assignment of nodeId can be application specific; typically by hashing an application specified value using SHA-1 [14]. The message keys are also application specific. For example, when inserting the credit of a node into the DHT (Section 2.4) the message key could be some identifier that uniquely identifies that node.

The Pastry overlay network is self-organizing, and each node maintains only a small routing table of $O(\log N)$ entries, where N is the number of nodes in the overlay. Each entry maps a nodeId to an IP address. Specifically, a Pastry node's routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $(2^b - 1)$ entries each. Each of the $(2^b - 1)$ entries at row n of the routing table refers to a node whose nodeId shares the first n digits with the present node's nodeId, but whose $(n+1)$th digit has one of the $(2^b - 1)$ possible values other than the $(n+1)$th digit in the present node's nodeId. Each node also maintains a *leafset*, which keeps track of its l immediate neighbors in the nodeId space. The leafset can be used to deal with new node arrivals, node failures, and node recoveries as explained below. Messages are routed by

Pastry to the destination node in $O(\log N)$ hops in the overlay network.

2.2.2 Discovering free cycles

While p2p overlay networks have been mainly used for data-centric applications, our system exploits p2p overlays for compute cycle sharing. Specifically, it organizes all the participating nodes into an overlay network, and uses the overlay to discover available compute cycles on remote nodes.

To discover nearby nodes that have free cycles, our system exploits the *locality-awareness* property of Pastry [7] to maintain and locate nearby available resources to dispatch jobs for remote execution. This property means that each entry in a Pastry node n's routing tables contains the node n_c that is close to n among all the nodes in the system whose nodeIds share the appropriate prefix that would place them in that entry of n.

Periodically, each node propagates its resource availability and characteristics to its neighbors in the proximity space. This is achieved by propagating the resource information to the nodes n' in each node n's Pastry routing table rows. Node n' also forwards the resource information according to a Time-to-Live (TTL) value associated with every message. The TTL is the maximum number of hops in the overlay between n and the nodes that receive its resource information. Hence, the resource information is propagated to neighboring nodes within TTL hops in the overlay. Since Pastry routing tables contain only nearby nodes, this "controlled flooding" will cause resource information to be spread among nearby nodes in the proximity space. Each node that receives such an announcement caches the information in the announcement for its local matchmaking between jobs and available resources.

To locate a remote node for job execution, a node queries nearby nodes with available resources as accumulated in its local knowledge, taking proximity and credit-worthiness into account. The actual remote execution of the program and subsequent I/O activities are performed with the remote node directly, and do not go through the overlay.

Our p2p-based cycle sharing system tolerates node/network failures as follows. When the node for remote execution fails, the submitting node discovers an alternative node for re-execution. The fault handling can be implemented in the runtime system and made transparent to the user program. This is an important advantage over the traditional client/server model, where failure of a server implies explicit reselection by the client. For reasons of failure resiliency (or malicious node detection) via comparison of results from multiple nodes, a node may create multiple identical computations for remote execution.

2.3 Safety and portability through Java VMs

One of the goals of this project is to allow cycle sharing with a minimum of human-based administrative overhead. This requires that jobs be accepted from users who have not been vouched for by some accrediting organization. This, in turn, requires that submitted applications not be able to damage the hosting machine. Java's sandboxing abilities fit in well with this model. If submitters have undergone some additional verification (i.e. they are a trusted user), digital signature based security mechanisms can be used to allow potentially more harmful code to be executed, for example code that accesses the host's file system.

Java portability across different software and hardware environments significantly lowers the barriers to machines joining the pool of users and resources available on the network. Java's portability is in part because byte-code is well defined. Another reason for its portability is that much of the functionality that is provided by system libraries, and is not part of languages like C++ and Fortran (e.g. thread libraries and sockets), is provided by well specified standard Java libraries. As a consequence, in practice Java's interfaces to system services, e.g., sockets, appear to be more portable than with C++ implementations.

These attributes allow submitting nodes to have a larger number of potential hosts to choose from, and increases the probability that a program will execute correctly on a remote host.

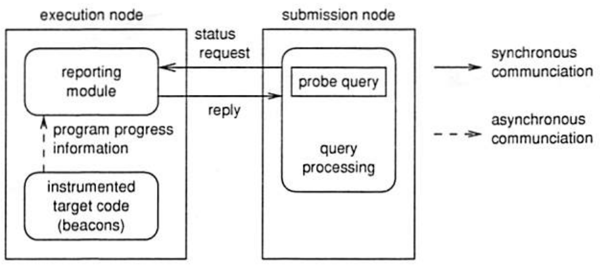

execution node
submission node
status request
reply
reporting module
probe query
query processing
program progress information
instrumented target code (beacons)
synchronous communciation
asynchronous communciation

Figure 2: Compiler instrumentation for monitoring job progress. The communication between the instrumented beacons and the reporting module is asynchronous, whereas it is synchronous between the querying node and reporting module.

2.4 Accountability through Java, p2p networks, and credit-worthiness

Accountability in our system is achieved through a system of credits. The storage and retrieval of these credits is accomplished through the p2p network, and the monitoring of long-running jobs to determine if credits should be issued is done using a JVM.

2.4.1 Compiler support for progress monitoring

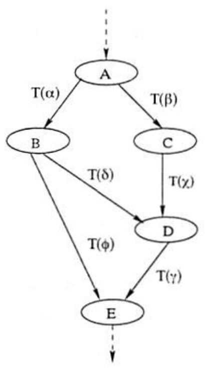
The basic idea behind the compiler assisted monitoring of the program execution is that the compiler will instrument the program with beacons which will periodically emit some indication of the program's progress. Because Java is a dynamically compiled language, commercial and research JVMs typically monitor the "hotness" of executing methods by either inserting sampling code that is periodically executed or executed on method entry, or by observing currently active methods. These techniques develop an approximation of the time spent in a method. Our system exploits this information to monitor the progress of the program and sends this information asynchronously to a separate program called the reporting module. The reporting module then buffers this information so that it can reply to queries from the program owner.

When a program owner queries the reporting module, the reporting module creates a progress report from the data it has so far collected, and replies to the program owner immediately. The owner can then process this report and determine whether its program is making progress. Figure 2 shows this setup. The advantage of having the reporting module is two-fold: (i) the application program does not have to suspend itself while waiting to be probed by the job owner, instead the data is buffered in the reporting module, and (ii) the design of the beacons is decoupled from the design of the queries. Moreover, the reporting module can run on the execution node, the submission node, or any other node that can communicate with both the execution and the submission node. For this reason, the reporting module is implemented as a separate process.

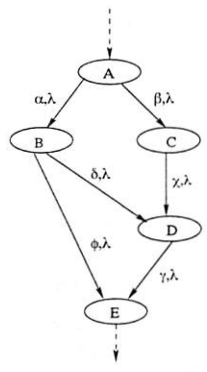
On a truly malicious, as opposed to overloaded or otherwise defective system, this simple beacon can be spoofed by the malicious system replaying old beacons. In the worst case, determining that a program on a remote system has run to completion and without tampering is as hard as actually running the program. It is our assumption in this project that we are not executing on truly malicious machines, rather we are running on machines that may be overcommitted, or that may be "fraudulently" selling cycles that don't exist in order to gain credits to purchase real cycles. Our goal is to uncover fraud and overcommitted nodes before the system is exploited "too heavily" by fraudulent or over-extended machines, not to prevent all fraud. Thus our system does not need to detect all fraudulent or overcommitted systems, but rather must allow fraudulent and overcommitted systems to be detected "soon enough". This is analogous to the goal of credit rating services in "real world" commerce, which is not to prevent any extension of credit to unworthy recipients, but rather to bound the extent to which they can receive credit to an amount that can be absorbed by the system. We stress that system is general enough to support either decentralized or centralized credit reporting mechanisms, depending on any legal requirements or requirements of the member nodes.

We are developing audit methods for better, albeit still not perfect, credit reporting. Figure 3(a) show a graphical representation of a program – it can be either a control flow graph or a calling graph. This graph can be treated as a transducer[1], or as a finite state automata (FSA) that accepts the language defined by the strings emitted by the transducer. In this model, the execution of the program corresponds to the transducer, with the reporting module implementing the recognizing FSA. The compiler will insert beacons to implement the

[1]A transducer is a finite state automata that emits information on state transitions.



(a) A flow or calling graph as a transducer



(b) The FSA corresponding to this transducer

Figure 3: A flow graph as a transducer and recognizing Finite State Machine.

transducer within the executing application and will also create the associated FSA.

Output emitted by the transducer can be simple, such as method names, or more complicated, such as method names, ranges of induction variables and the values of the induction variables themselves. However, this system may still not be secure since either another program understanding tool or a human could reverse-engineer the transducer in the application program and use that information to spoof the reporting module.

2.4.2 Issuing credits and assessing credit-worthiness

To ensure the compensation of consumed cycles on node *B* by node *A*, we propose a distributed credit-based mechanism. There are two building blocks of our approach: (i) credit-reports which are entities that can be "traded" in exchange for resources, and (ii) a distributed feedback system which provides the resource contributors with the capability to check the credit history of a node, as well as to submit feedback about the behavior of a node. For simplicity, we assume that all jobs are equivalent in terms of the amount of resources they consume.

The distributed feedback database is built on top of the Distributed Hash Table (DHT) supported by the underlying structured p2p overlay. It maintains the feedback for each node regarding its behavior towards honoring credits. Any node in the system can access this information and decide whether to allow an exchange with a requesting node, or consider it a rogue node and avoid any dealings with it. In this way, a node can individually decide to punish a node whose consumption of shared resources has exceeded its contribution to other nodes by some threshold determined by the deciding node.

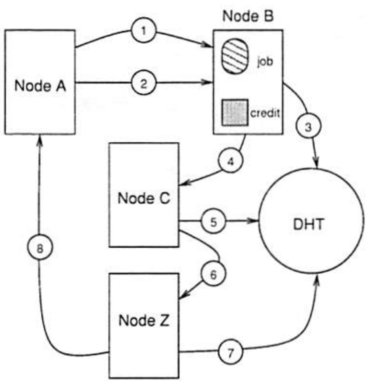


Figure 4: Various steps to ensure proper compensation of a contributed resource.

Figure 4 shows the various steps involved in ensuring that *B* is adequately compensated for its contribution. When A runs a job on B (1 in Figure 4), A will issue a (digitally signed) credit to B (2 in the Figure) (This credit is similar to the claim in Samsara [8]; it can be "traded" with other nodes for exchange of equivalent resources). A will give the credit a unique sequence number – unique in that A will issue no other credits with that number. B will digitally sign the credit and hash it into a repository in the p2p network (3 in the Figure). The credit hash is generated by hashing a function of A and the sequence number.

If B gives the credit to C (4 in the Figure), B will digitally sign the credit before giving it to C. C will digitally sign it, compute the hash based on (A, unique number) and store it (5 in the Figure). The storing node will replace the existing copy of the credit with the new copy. It knows it can do this since the end of the signing sequence is "B, B, C", i.e. the last-1 and last-2 signatures match, showing that the last-1 signer is the previous owner and is allowed to transfer the certificate.

If the certificate goes to Z (6 and 7 in the Figure) and returns to A (8 in the Figure), A destroys it. Since the end of the signing sequence is "Z, Z, A", the system knows that the transfer to A is valid and A is the owner, and therefore A can choose to have the credit destroyed. Note that this also allows credits to be destroyed by any owner (i.e. C above could have asked that the credit be destroyed, not saved) perhaps because of monetary payments, lawsuits, bankruptcy of the root signer, etc.

Because a credit hashes to a fixed location, attempts to forge credits (for example in a replay attack) will leave multiple copies of the credit (identified by its unique sequence number) in the same DHT location, and the second forged credit will not be saved. Thus if B tries to give the same credit d to both C and Z, one would be rejected (say Z's) and Z could then refuse to run B's job. Should a malicious node save both credits, a node checking on the credit worthiness of the issuing node can determine that there are two credits with the same number, and either ignore or factor this into its evaluation of both the issuing node *and* node B.

The feedback information is used to enforce contribution from selfish nodes as follows. In Figure 4, before node B executes a job on behalf of node A, it retrieves all the feedback for A from the DHT, verifies the signatures to ensure validity, and can decide to punish A by refusing its job if A's number of failures to honor credits has exceeded some threshold determined by B. We note that this system allows independent credit rating services to be developed that a submitting node can rely on for evaluating the credit-worthiness of a host.

2.4.3 Potential threats and vulnerabilities

It is our assumption in this project that we are not executing on truly malicious machines, rather we are running on machines that may be overcommitted, or that may be "fraudulently" selling cycles that don't exist in order to gain credits to purchase real cycles. Thus we limit out discussion of potential threats to be from such misbehaving nodes in the following.

The first scenario deals with the timing between the issuing of credits and the running of submitted jobs. In particular, a running node may refuse to spend further cycles upon receiving credits from job submitting nodes, and conversely, the submitting node may refuse to issue credits upon hearing the completion of its remote job execution. To provide mutual assurance, we propose the use of incremental credit issuing. Consider the case when A is submitting a job to run on B. Under the incremental credit issuing scheme, A gives incremental credits when it sees progress of its job on B. A might also choose to checkpoint its job when issuing an incremental credit to eliminate the chance of losing work that has been paid for. Secondly, A can also issue a negative feedback for B if it misbehaves. This is done as follows. A monitors its job on B at predefined intervals. On each interval that A finds its job stalled, it calculates a probability $\mathtt{q}$, which increases exponentially with increasing number of consecutive failures of B to allow the job to progress. A then issues a negative feedback for B with the probability $\mathtt{q}$. This scheme allows B to have transient failures, but punishes it for chronically cheating. Conversely, the case where A refuses to honor its issued credit is already addressed by the distributed feedback mechanism.

The next scenario results from the fact that if each feedback is inserted in the DHT only once, even though DHT replicates the credit among nodes nearby in the nodeId space, if the node storing the main replica acts maliciously, the credit information may not be retrieved correctly. To overcome this, we purpose inserting each credit into the DHT multiple times by making use of a predetermined sequence of salts known to all the participants. For each known salt, a credit hash is generated by hashing the concatenated function of the issuer's nodeId, the sequence number, and the salt. The signed credit is then inserted into the DHT. In this way, each credit will be available at multiple mutually-unaware nodes. When a participant intends to verify the integrity of another participant, it can choose to retrieve multiple copies of the credit instead of one. The number it attempts to retrieve can vary between one and the maximum copies that are allowed to be inserted into the system. Once these copies are retrieved, the node can compare them for trustworthiness. In case of mismatch, a majority vote can be adopted and the version of the credit that has the highest number of occurrence is assumed to be the trusted one. This potentially increases the amount of data that is inserted and retrieved from the DHT. However, the size of credit is usually small, and the additional benefits of having multiple insertions outweigh the increase in the message overhead.

Another problem may arise where the credits are never "traded" back to the issuing node. This does not indicate misbehavior, but can result in a large number of un-utilized credits accumulated in the system. Therefore, where possible, a node tries to "trade" a remote credit it holds, rather than issue its own credit. Moreover, each credit-report has a timestamp, and nodes can determine how old a

Module name	Functionality
`announceC`	Creates resource information announcements and sends them to the neighboring nodes.
`dbaseC`	Manages the local knowledge base on a node.
`matchmakerC`	Finds suitable nodes for requested job runs from available remote nodes.
`execC`	Sets up the execution environment for a job on a matched node and initiates the execution.

Table 1: Modules in the prototype implementation and their functions.

credit is. In case the age of a credit-report originated from A increases more than a system-wide threshold, the credit holding node will try to exchange it with A first, before trying to locate some other resources in the network. This will help in reducing the number of credit-reports in the system. Intra-organizational networks can clear the system periodically using internal budgeting procedures. We note that the load imposed on the system by large amounts of circulating credit is much less than in systems like Samsara, where credit takes the form of physical disk space held hostage, and consequently reduces the amount of system resources available for other purposes.

3 Implementation

We have built a prototype of our proposed scheme for p2p based resource discovery and accountability using Java 1.4.2 API specification. We utilize the Pastry [32] API for p2p functionality, and PAST [31] for storing the distributed feedback in a fault-tolerant and distributed manner. The implementation is done on nodes with Pentium IV 2 GHz processors, 512 MB RAM, running Linux kernel 2.4.18, connected via 100 Mb/s Ethernet. The prototype can be divided into various software modules as listed in Table 1.

An interesting observation in the implementation process was the duplicated reception of the same node's resource announcements at a node because a node may be on the routing tables of multiple remote nodes. To prevent duplicate messages from flooding the system, each node assigns a 32-bit sequence number to its announcements. The number is chosen at random at the start of a node, and increases monotonically for the life of the node. When an announcement is received, its sequence is first compared with the sequence number of the last message received from the originator. A sequence number with a value equal to or less than the last seen message implies that the message is a duplicate and can be discarded.

Our execution node uses an augmented version of the Jikes RVM [1], running on adaptive configuration. In the adaptive system, there exists an instrumentation framework that increments method invocation counters as the application proceeds. The counters are then stored in a database whose contents are examined by the controller thread to help make recompilation decisions. Since the application progress reports that we require can be inferred from this database, we do not have to add inline code into the application to determine it. Instead, we augment the system with a progress monitoring thread. In other words, the beacon is implemented as a thread that periodically looks into the database and sends the method invocation counter values to the reporting module, together with the timestamp. This information serves as an indicator for job progress. The interval between successive reports is a parameter that is specified when remote jobs are submitted. We will examine the effect of this interval on the execution overhead in our experimentation.

In our implementation the reporting module is located on the same host as the execution node. This enables the reporting module to record and report the system load information on the execution node as well, which provides further grounds for the probing module on the submission node to determine whether the remote host is cheating. (If the host is heavily loaded, we might assume the host is not cheating even though the progress is reasonably small). Also, hosting the reporting module on the same node incurs additional overhead, and we are able to study its effects through experimentation.

Communications between the execution node and the reporting module, and between the reporting module and the probing module, are implemented through UDP packets. We choose to use socket-based communication because it enables us to move the reporting module off the execution node if

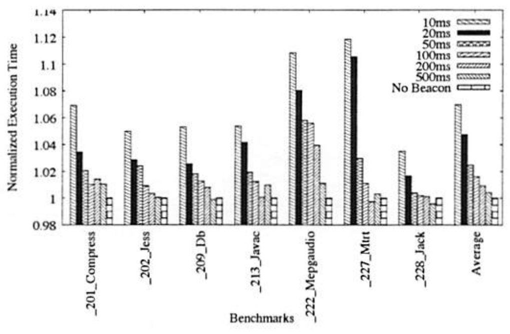


Figure 5: Overhead of beacons with a remote reporting module.

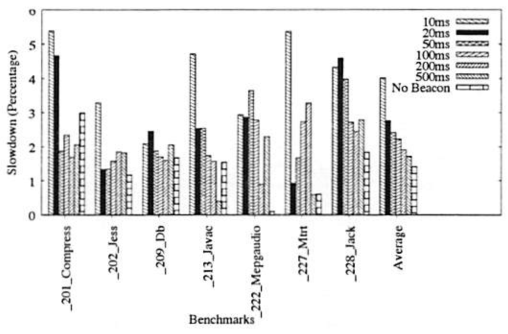


Figure 6: Overhead of hosting the reporting module on the execution node.

needed, and UDP is preferred because it has a lower overhead. As we are only interested in getting an indicator of application progress, losing some packets occasionally is not a big problem.

4 Evaluation

This section gives experimental results gathered by executing a modified version of the Jikes RVM on hardware, and by simulation studies of a p2p overlay network.

4.1 Instrumentation overhead

In this section, we determine the effect of progress monitoring on application performance by studying the overheads of the beacons and the reporting module.

To study the overhead of beacons alone, we first run the reporting module on a different node than the execution node. Figure 5 shows the overhead of adding the beacon thread in the virtual machine. Experiments were run with the SpecJvm98 benchmark suite on data sizes of 100. Instrumentation results were reported to the remote reporting module at time intervals of 10, 20, 50, 100, 200 and 500 milliseconds, respectively. We observe that on average, when the time interval between successive reports is 500ms, the performance impact is less than 1%. For an actual system, reporting intervals would be in minutes or seconds, so the overhead on the program performance would be effectively zero. Notice that this is the overhead of monitoring the program progress only. The cost of instrumentation in the Jikes RVM system, which we are leveraging for our technique, is about 6.3% [3], and can be lowered with larger instrumentation intervals.

Next we determine the overhead of hosting the reporting module on the execution node. We compare the performance of this setup with one that uses a remote reporting module (as in the previous case). Figure 6 shows the results. There are three sources for the overhead: network traffic to and from the reporting module, cost of periodically collecting host system load information, and overhead of maintaining the application progress and system load database. Application execution on the host slows down as a result of competition for CPU cycles. The first and last kinds of the overhead will grow linearly with the number of remotely executing jobs and they also grow as instrumentation and probing frequencies increase. The cost of system load information is proportional to the interval of load probing. In our experiment we assume one running VM, one probing node, and a load test interval of 200ms. We get the overhead for the job running on the VM. We did not do tests with multiple VMs on the same execution node because in that case the overhead is difficult to express in terms of the slowdown, and multiple working VMs compete for CPU cycles among themselves. The presence of the reporting module, without beacons, causes a slowdown of 1.4%. Increased reporting density incurs additional slowdowns, but again with realistic reporting frequencies, we can assume that part of the overhead to be zero.

4.2 Simulation results

In this section, we evaluate the effectiveness of our p2p scheme on discovering appropriate resources for execution of jobs and on enforcing fairness in cycle

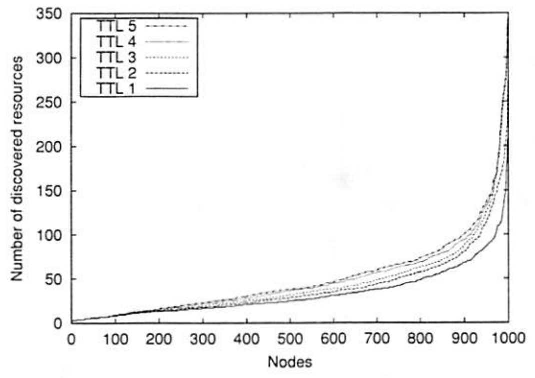


Figure 7: The number of resources available at each node with increasing TTL. The nodes are sorted in increasing order of discovered resources.

sharing by simulating a network of 1000 nodes. We developed a simulator of our proposed system on top of Pastry running with the direct communication driver, which allows the creation of multiple Pastry nodes on a single machine. We used the transit-stub Internet model [39] to generate an IP network with 10 stub domain routers, 100 transit domain routers, and attach 1000 nodes randomly to the 100 stub domain routers.

In the first set of simulations, we measure the effectiveness of our protocol for disseminating resource announcements. We assume each node has some resource to announce to the network, and uses the defined protocol to send the announcements. We run the simulation five times with TTL varying from 1 to 5. For each simulation, we measure the number of remote nodes whose announcements reach any given node. Figure 7 shows how the number of "discovered" nodes by each node increases with the TTL value. Increasing TTL results in more resources being discovered, but it also implies that resources may be far away in the network proximity space.

To measure the effectiveness of our credit-based scheme for enforcing fairness, we simulate cycle sharing among the 1000 nodes, with 500 nodes actively submitting jobs, while the rest of the nodes only contribute cycles. The TTL is fixed at three for this set of observations. Each of the 500 submitting nodes is fed with a randomly generated sequence of 100 jobs of equal length, each running for nine time units. The interarrival time between the jobs follows a uniform distribution between 1 and 17 time units. We compare the job throughput of three scenarios: (1) all 500 submitting nodes are honest; (2) 250 submitting nodes are honest and 250 submitting nodes cheat by only submitting jobs and never accepting remote jobs, and the fairness mechanism is not turned on; and (3) same as (2) but with the fairness mechanism turned on. For case (3), the threshold for detecting cheating nodes is set to three, i.e., a cheating node can run up to three jobs without compensating the system, but when it attempts to run more jobs, other nodes ignore its requests for resources. The job throughput under these scenarios are shown in Figures 8, 9, and 10, respectively.

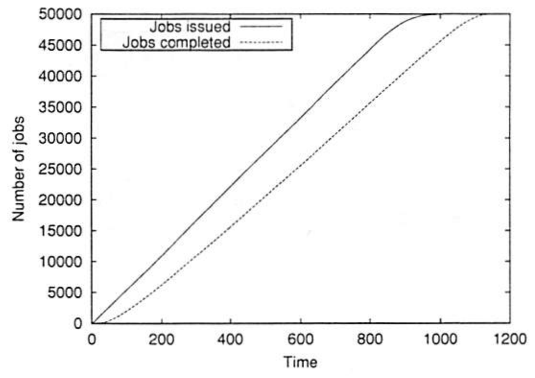


Figure 8: The number of jobs issued, and completed over the period of our trace. Every node contributes resources to the system and is fair.

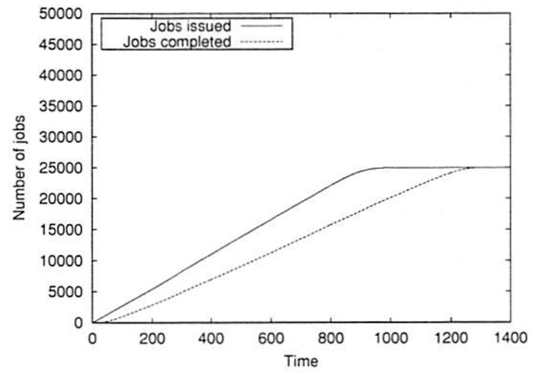


Figure 9: The number of jobs issued, and completed over the period of our trace for the non-cheating nodes. The feedback system is disabled.

Figure 8 shows the number of jobs issued and the number of jobs completed against our simulated time when all 500 nodes are honest. It is observed that the jobs do not have to wait if free resources are available. The number of completed jobs closely follow the number of issued jobs. The slight increase in the difference over time is due to the fact the more jobs were requested than the available resources. However, all jobs completed at about the 1150th time unit, only 150 time units after all the jobs were issued.

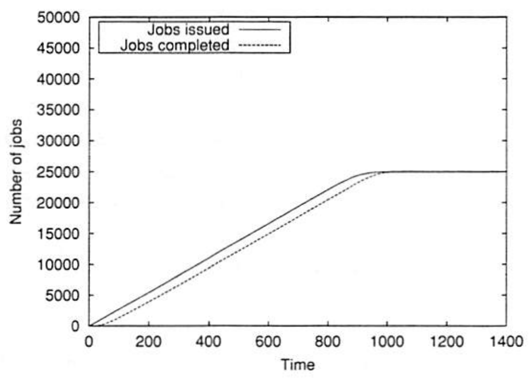


Figure 10: The number of jobs issued, and completed over the period of our trace for the non-cheating nodes. The feedback system is enabled.

Figure 9 shows when 250 submitting nodes cheat and the credit-based mechanism is not turned on, the non-cheating nodes experience a larger delay in their job completion. There is a delay of 345 time units for all the jobs to complete. This is significant considering that the job lengths are only 9 time units.

Finally, Figure 10 shows the credit-based mechanism effectively isolates the cheating nodes and the jobs from the non-cheating nodes made more progress compared to the case without the credit-based mechanism. Note that the job delay in this case is less than that in Figure 8, because here 250 non-cheating nodes were sharing cycles of a total of 750 nodes, where as in the latter case 500 nodes shared cycles of 1000 nodes. This simulation shows that the credit-based mechanism quickly prohibits cheating nodes from consuming other nodes' cycles.

5 Related work

We discuss related work on cycle sharing over the Internet, on compiler instrumentations, and on fairness in p2p storage sharing systems.

Cycle sharing over the Internet The idea of cycle sharing among a large number of administratively independent, geographically dispersed, off-the-shelf desktops is popularized by the SETI@home [34] project. Similar approaches for solving large scale scientific problems are also adopted in systems such as Distributed.NET [10], Entropia [12], Genome@home [18] , and Nile [28] . These systems implement a central manager that is responsible for the distribution of the problem set, and the collection and analysis of the results. Users typically download the client programs manually and then execute them on their resources. The client programs are specially developed applications that the resource owners have to explicitly trust [6]. The clients periodically contact the central managers to provide results and to receive further data for processing. The clients are pure volunteers in nature, i.e., they do not receive any resource contribution for their own tasks. The aim of our project is to provide all nodes in the system with the capability to utilize shared resources. This provides an incentive for more resource owners to contribute resources to the system, hence increasing the instantaneous compute capacity of the system.

Various grid platforms also share the same goal of distributed sharing resources. Condor [24] provides a mechanism for sharing resources in a single administrative domain by harnessing the idle-cycles on desktop machines. Globus [15] and Legion [20] allow users to share resources across administrative domains. However, the resource management is hierarchical, and the users have to obtain accounts on all the resources that they intend to use [6]. PUNCH [23] decouples the shared resource users from the underlying operating system users on each resource, hence eliminating the need for accounts on all the shared resources. Sun Grid Engine [37] is another system that harnesses the compute powers of distributed resources to solve large scale scientific problems. However, all of these systems rely on some forms of centralized resource management and therefore are susceptible to performance bottlenecks, single-point of failures, and unfairness issues that our system avoids by using p2p mechanisms.

Fair peer-to-peer storage sharing The idea of enforcing fairness has been extensively studied in peer-to-peer storage systems, motivated by the usage studies [11, 33] which show that many users consume resources but do not compensate by contributing. CFS [9] allows only a specified storage quota for use by other nodes without any consideration for the space contributed to the system by the consumer. PAST [31] employs a scheme where a trusted third party holds usage certificates that can be used in determining quotas for remote consumers. The quotas can be adjusted according to the contribution of a node. Samsara [8] enforces fairness in p2p storage without requiring trusted third parties, symmetric storage relationships, monetary payment, or certified identities. It utilizes an extensive claim management which leverages selfish behavior of each node to achieve an overall fair system. The fairness in our cycle sharing system was motivated by Samsara. However, fairness in cycle sharing is more complex than in data sharing as once a computation is completed, the execution node has no direct means of punishing a cheating consumer. SHARP [17] provides a mechanism for resource peering based on the exchange of *tickets* and *leases*, which can be traded among peering nodes for resource reservation and committed consumption. Credits in our system are similar to *tickets* in SHARP. However, our system uses the credit reports to enforce fairness of sharing, and not as a mean for advance resource reservations.

There have also been efforts to design a general framework for trading resources in p2p systems. *Data trading* [5] is proposed to allow a consumer and a resource provider exchange an equal amount of data, and cheaters can be punished by withholding the data. The approach requires symmetric relationships and do not apply well to p2p systems where there is very little symmetry in resource sharing relationships. The use of micropayments as incentives for fair sharing is proposed in [19]. Fileteller [22] suggests the use of such micropayments to account for resource consumption and contribution. In [38], a distributed accounting framework is described, where each node maintains a signed record of every data object it stores directly on itself or on other nodes on its behalf, and each node periodically audits random other nodes by comparing multiple copies of the same records. The system requires certified entities to prevent against malicious accusations, and the auditor has to work for other nodes, without any direct benefit. Our system implements a distributed accounting system as well, where a node verifies credit reports of a remote node only when it has to do an exchange with it, which is a direct benefit.

Compiler instrumentation and proof carrying code The concept of compiler generated instrumentation and monitoring of program execution within a JVM is not new. The Sun Hotspot [29] compiler, compilers for the IBM JDK [36], and compilers for the Jikes RVM [1] all use either compiler generated program instrumentation or an examination of the active routines on the stack to determine *hot* methods. This in turn is similar to profiling [2, 4, 13] for collecting information about where time is spent during a program's execution. These efforts are orthogonal to our work, and could complement our work by providing a mechanism for collecting information about total program run time. Our novelty is in bootstrapping off the already existing monitoring of program executions supported by JVMs as part of their optimization strategy to allow the progress of a program to be remotely tracked. Other projects (e.g. [21]) have used instrumentation to collect data for performance purposes, and allows on-the-fly instrumentation of statically compiled programs.

The GRAM component of the Globus project [16] also monitors program execution, and uses this information to change the resource requirements of the application to give better quality of service. Our work differs in our motivation – we are using the monitoring to determine if we are getting a resource as promised; and in our implementation – the monitoring measures program performance via automatic instrumentation by a JVM and not by external measures or by programmer inserted callouts from the program. This allows us to not require individual programs to be adapted to our system in order to use it.

The ultimate goal of our work is related to proof carrying code [27]. We differ from that work in that the ultimate goal here is to have a program certify to the submitter, rather than the host, facts about its execution, with these facts bound to the program form itself.

6 Conclusion

We have described the design of a system, and our implemented prototype, that exploits the safety, portability and internal profiling capabilities of Java and Java Virtual Machines. This system allows a decentralized p2p network to be used to advertise and allocate resources, and contains a credit system that allows decentralized sharing of resources and evaluation of credit-worthiness. Our experimental results show that our fairness mechanisms work well to punish cheating nodes, and our monitoring of program progress has effectively zero overhead. Because of its decentralized nature, leading to low costs for entry and exit from the network, our system is ideal for constructing ad-hoc intra-organizational networks of pooled resources, and for constructing pools of resources for small business and educational purposes.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was supported in part by NSF CAREER award grant ACI-0238379 and NSF grants CCR-0313026 and CCR-0313033.

References

[1] B. Alpern and *al. et.* The Jalapeo Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 1997.

[3] M. Arnold. *Online Profiling and Feedback-Directed Optimization of Java.* PhD thesis, Rutgers, The State University of New Jersey, October 2002.

[4] T. Ball and J. R. Laurus. Efficient path profiling. In *Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] B.F. Cooper and H. Garcia-Molina. Peer-to-peer resource trading in a reliable distributed system. In *Proc. First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.

[6] A. R. Butt, S. Adabala, N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Grid-computing portals and security issues. *Journal of Parallel and Distributed Computing: Special issue on Scalable Web Services and Architecture*, 63(10):1006–1014, October 2003.

[7] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002, 2002. ⟨ http://research.microsoft.com/~antr/PAST/ localtion.ps ⟩ (17 Oct 2003).

[8] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, October 2001.

[10] distributed.net. distributed.net projects (11 April 2003). ⟨ http://www.distributed.net/projects.php ⟩ (28 September 2003).

[11] E. Adar and B.A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.

[12] Entropia, Inc. Entropia: Pc grid computing (16 June 2003). ⟨ http://www.entropia.com/index.asp ⟩ (28 September 2003).

[13] J. Fenlason and R. Stallman. GNU gprof manual (Nov 7, 1998). ⟨ http://www.gnu.org/manual/ gprof-2.9.1/gprof.html ⟩ (Oct 14, 2003).

[14] FIPS 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington D.C., April 1995.

[15] I. Foster and C. Kesselmann. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, Jan. 1997.

[16] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Proc. 8th International Workshop on Quality of Service*, 2000.

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

[18] Genome@home. Genome at home(26 September 2003). ⟨ http://www.stanford.edu/group/pandegroup/ genome/index.html ⟩ (29 September 2003).

[19] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proc. Third ACM Conference on Electroni Commerce*, Tampa, FL, 2001.

[20] A. S. Grimshaw and W. A. Wulf. Legion – A View from 50,000 feet. In *Proc. 5th IEEE International Symposium on High Performance Distributed Computing(HPDC'96)*, Syracuse, NY, 1996.

[21] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proc. IEEE PACT*, pages 201–, 1997.

[22] J. Ioannidis, A. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proc. Sixth Annual Conference on Financial Cryptography*, Bermuda, 2002.

[23] N. H. Kapadia and J. A. B. Fortes. PUNCH: An architecture for Web-enabled wide-area network-computing. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2(2):153–164, Sep. 1999.

[24] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, pages 104–111, San Jose, CA, 1988.

[25] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The NINJA project: Making Java work for high performance computing. *Communications of the ACM*, 44(10):102–109, October 2001.

[26] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000. IBM Research Report RC 21166.

[27] G. Necula. Proof carrying code. 1997.

[28] Nile. Scalable solution for distributed processing of independant data (18 June 1999). ⟨ http://www.nile.cornell.edu/index.html ⟩ (29 September 2003).

[29] M. Paleczny, C. Click, and C. Vick. The Java HotSpot server compiler. In *Proc. 2001 USENIX Java Virtual Machine Symposium*, 2001.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 161–172, San Diego, CA, 2001.

[31] A. Rowstron and P. Druschel. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles*, October 2001.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[33] S. Saroiu, G. Krishna, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. SPIE Conference on Multimedia Computing and Networking*, San Jose, CA, 2002.

[34] SETI@home. Search for extraterrestrial intelligence at home (29 September 2003). ⟨ http://setiathome.ssl.berkeley.edu/index.html ⟩ (29 September 2003).

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 149–160, San Diego, CA, 2001.

[36] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 180–195, 2001.

[37] Sun(TM) Microsystems. Sun ONE Grid Engine Software (26 June 2003). ⟨ http://wwws.sun.com/software/gridware/sge.html ⟩ (29 September 2003).

[38] T-W.J. Ngan and D.S. Wallach and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, 2003.

[39] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE INFOCOM*, March 1996.

[40] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Towards Virtual Networks for Virtual Machine Grid Computing

Ananth I. Sundararaj Peter A. Dinda
{ais,pdinda}@cs.northwestern.edu
Department of Computer Science, Northwestern University

Abstract

Virtual machines can greatly simplify wide-area distributed computing by lowering the level of abstraction to the benefit of both resource providers and users. Networking, however, can be a challenge because remote sites are loath to provide connectivity to any machine attached to the site network by outsiders. In response, we have developed a simple and efficient layer two virtual network tool that in effect connects the virtual machine to the home network of the user, making the connectivity problem identical to that faced by the user when connecting any new machine to his own network. We describe this tool and evaluate its performance in LAN and WAN environments. Next, we describe our plans to enhance it to become an adaptive virtual network that will dynamically modify its topology and routing rules in response to the offered traffic load of the virtual machines it supports and to the load of the underlying network. We formalize the adaptation problem induced by this scheme and take initial steps to solving it. The virtual network will also be able to use underlying resource reservation mechanisms on behalf of virtual machines. Both adaptation and reservation will work with existing, unmodified applications and operating systems.

1 Introduction

Recently, interest in using OS-level virtual machines as the abstraction for grid computing and for distributed computing in general has been growing [11, 21, 13, 15]. Virtual machine monitors such as VMware [37], IBM's VM [17], and Microsoft's Virtual Server [27], as well as virtual server technology such as UML [4], Ensim [9], and Virtuozzo [36], have the potential to greatly simplify management from the perspective of resource owners and to provide great flexibility to resource users. Much grid middleware and application software is quite complex. Being able to package a working virtual machine image that contains the correct operating system, libraries, middleware, and application can make it much easier to deploy something new, using relatively simple middleware that knows only about virtual machines. We have made a detailed case for grid computing on virtual machines in a previous paper [11].

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, EIA-0224449, and a gift from VMWare. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Unlike traditional units of work in distributed systems, such as jobs, processes, or RPC calls, a virtual machine has, and must have, a direct presence on the network at layer 3 and below. We must be able to communicate with it. VMM software recognizes this need and typically creates a virtual Ethernet card for the guest operating system to use. This virtual card is then emulated using the physical network card in the host machine in one of several ways. The most flexible of these bridges the virtual card directly to the same network as the physical card, making the virtual machine a first class citizen on the same network, indistinguishable from a physical machine.

Within a single site, this works very well, as there are existing mechanisms to provide new machines with access. Grid computing, however, is intrinsically about using multiple sites, with different network management and security philosophies, often spread over the wide area [12]. Running a virtual machine on a remote site is equivalent to visiting the site and connecting a new machine. The nature of the network presence (active Ethernet port, traffic not blocked, routable IP address, forwarding of its packets through firewalls, etc) the machine gets, or whether it gets a presence at all, depends completely on the policy of the site. The impact of this variation is further exacerbated as the number of sites is increased, and if we permit virtual machines to migrate from site to site.

To deal with this problem in our own project, we have developed VNET, a simple layer 2 virtual network tool. Using VNET, virtual machines have no network presence at all on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from one network to another. For example, all of a user's virtual machines can be made to appear to be connected to the user's own network, where the user can use his existing mechanisms to assure that they have appropriate network presence. Because the virtual network is a layer 2 one, a machine can be migrated from site to site without changing its presence—it always keeps the same IP address, routes, etc. The first part of this paper describes how VNET works and presents performance results for local-area and wide-area use. VNET is publicly available from us.

As we have developed VNET, we have come to believe that virtual networks designed specifically for virtual machine grid computing can be used for much more than simplifying the management problem. In particular, because they see all of the traffic of the virtual machines, they are in an ideal position to (1) measure the traffic load and application topology of the virtual machines, (2) monitor the underlying network, (3) adapt application as measured by (1) to the network as measured by (2) by relocating virtual machines and modifying the virtual network topology and routing rules, and (4) take advantage of resource reservation mechanisms in the underlying network. Best of all, these services can be done on behalf of existing, unmodified applications and operating systems running in the virtual machines. The second part of this paper lays out this argument, formalizes the adaptation problem, and takes initial steps to solving it.

2 Related work

Our work builds on operating-system level virtual machines, of which there are essentially two kinds. Virtual machine monitors, such as VMware [37], IBM's VM [17], and Microsoft's Virtual Server [27] present an abstraction that is identical to a physical machine. For example, VMWare, which we use, provides the abstraction of an Intel IA32-based PC (including one or more processors, memory, IDE or SCSI disk controllers, disks, network interface cards, video card, BIOS, etc.) On top of this abstraction, almost any existing PC operating system and its applications can be installed and run. The overhead of this emulation can be made to be quite low [33, 11]. Our work is also applicable to virtual server technology such as UML [4], Ensim [9], Denali [38], and Virtuozzo [36]. Here, existing operating systems are extended to provide a notion of server id (or protection domain) along with process id. Each OS call is then evaluated in the context of the server id of the calling process, giving the illusion that the processes associated with a particular server id are the only processes in the OS and providing root privileges that are effective only within that protection domain. In both cases, the virtual machine has the illusion of having network adaptors that it can use as it sees fit, which is the essential requirement of our work.

The Stanford Collective is seeking to create a compute utility in which "virtual appliances" (virtual machines with task-specialized operating systems and applications that are intended to be easy to maintain) can be run in a trusted environment [30, 13]. Part of the Collective middleware is able to create "virtual appliance networks" (VANs), which essentially tie a group of virtual appliances to an Ethernet VLAN. Our work is similar in that we also, in effect, tie a group of virtual machines together as a LAN. However, we differ in that the collective middleware attempts also to solve IP address and routing, while we remain completely at layer 2 and push this administration problem back to the user's site. Another difference is that we expect to be running in a wide area environment in which remote sites are not under our administrative control. Hence, we make the administrative requirements at the remote site extremely simple and focused almost entirely on the machine that will host the virtual machine. Finally, because the nature of the applications and networking hardware in grid computing tend to be different (parallel scientific applications running on clusters with very high speed wide area networks) from virtual appliances, the nature of the adaptation problems and the exploitation of resource reservations made possible by VNET are also different. A contribution of this paper is to describe these problems. However, we do point out that one adaptation mechanism that we plan to use, migration, has been extensively studied by the Collective group [31].

Perhaps closest to our work is that of Purdue's SODA project, which aims to build a service-on-demand grid infrastructure based on virtual server technology [21] and virtual networking [22]. Similar to VANs in the Collective, the SODA virtual network, VIOLIN, allows for the dynamic setup of an arbitrary private layer 2 and layer 3 virtual network among virtual servers. In contrast, VNET works entirely at layer 2 and with the more general virtual machine monitor model. Furthermore, our model has been much more strongly motivated by the need to deal with unfriendly administrative policies at remote sites and to perform adaptation and exploit resource reservations, as we describe later. This paper also includes detailed performance results for VNET, which are not currently available, to the best of our knowledge, for VAN or VIOLIN.

VNET is a virtual private network (VPN [10, 14, 19]) that implements a virtual local area network (VLAN [18]) spread over a wide area using layer 2 tunneling [35]. We are extending VNET to act as an adaptive overlay network [1, 3, 16, 20] for virtual machines as opposed to for specific applications. The adaptation problems introduced are in some ways generalizations (because we have control over machine location as well as the overlay topology and routing) of the problems encountered in the design of and routing on overlays [32]. There is also a strong connection to parallel task graph mapping problems [2, 23].

3 Virtuoso model

We are developing middleware, Virtuoso, for virtual machine grid computing that for a user very closely emulates the existing process of buying, configuring, and using an Intel-based computer, a process with which many users and certainly all system administrators are familiar with.

In our model, the user visits a web site, much like the web site of Dell or IBM or any other company that sells Intel-based computers. The site allows him to specify the hardware and software configuration of a computer and its performance requirements, and then order one or more of them. The user receives a reference to the virtual machine which he can then use to start, stop, reset, and clone the machine. The system presents the illusion that the virtual machine is right next to the user. The console display is sent back to the user's machine, the CD-ROM is proxied to the user's machine's CD-ROM, and the virtual machine appears to be plugged into the network side-by-side with the user's machine. The user can then install additional software, including operating systems. The system is permitted to move the virtual machine from site to site to optimize its performance or cost, but must preserve the illusion.

We use VMWare GSX Server [37] running on Linux as our virtual machine monitor. Although GSX provides a fast remote console, we use VNC [29] in order to remain independent of the underlying virtual machine monitor. We proxy CD-ROM devices using Linux's extended network block device, or by using CD image files. Network proxying is done using VNET, as described in the next section.

4 VNET: A simple layer 2 virtual network

VNET is the part of our system that creates and maintains the networking illusion, that the user's virtual machines are on the user's local area network. It is a simple proxying scheme that works entirely at user level. The primary dependence it has on the virtual machine monitor is that there must be a mechanism to extract the raw Ethernet packets sent by the virtual network card, and a mechanism to inject raw Ethernet packets into the virtual card. The specific mechanisms we use are packet filters, packet sockets, and VMWare's host-only networking interface. In the following, we describe VMWare's model of networking, how we build upon it, the interface of VNET, and performance results in the local and wide area.

We use the following terminology. The *User* is the owner of the virtual machines (his *VMs*) which he accesses using his *Client* machine. The user also has a *Proxy* machine for networking, although the Proxy and Client can be the same machine. Each VM runs on a *Host*, and multiple VMs may run on each Host. The *Local* environment of a VM is the LAN to which its Host it is connected, while the *Remote* environment is the LAN to which the Client and the Proxy are connected.

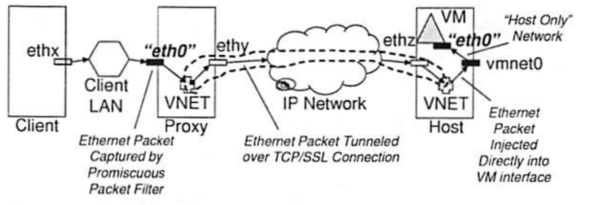
4.1 VMWare networking

VMWare, in its Workstation and GSX Server variants, can connect the virtual network interface to the network in three different ways. To the operating system running in the virtual machine (the VM), they all look the same. By themselves, these connection types are not well suited for use in a wide-area, multi-site environment, as we describe below.

The simplest connection is "bridged", meaning that VMWare uses the physical interface of the Host to directly emulate the virtual interface in the VM. This emulation is not visible to programs running on the Host. With a bridged connection, the VM shows up as another machine on the Local environment, the LAN of the Host. This creates a network management problem for the Local environment (What is this new machine that has suddenly appeared?) and for the User (Will this machine be given network connectivity? How? What's its address? Can I route to it?). Furthermore, if the VM is moved to a Host on a different network, the problems recur, and new ones rear their ugly head (Has the address to the VM changed? What about all its open connections and related state?)

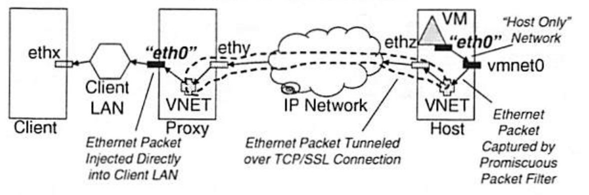
The next form of connection is the host-only connection. Here, a virtual interface is created on the Host which is connected to the virtual interface in the VM. When brought up with the appropriate private IP addresses and routes, this enables programs on the host to talk to programs on the VM. Because we need to be able to talk to the VM from the Client and other machines, host-only networking is insufficient. However, it also has the minimum possible interaction with network administration in the Local environment.

The final form of connection is via network address translation (NAT), a commonly used technique in border routers and firewalls [7]. Similar to a host-only connection, a virtual interface on the Host is connected to the virtual interface on the VM, and appropriate private IP addresses and routes are assigned. In addition, a daemon running on the Host receives IP packets on the interface. For each outgoing TCP connection establishment (SYN), it rewrites the packet to appear to come from the IP address of the Host's regular interface, from some unused port. It records this mapping from the IP address and port on the VM to the address and port it assigned. Mappings can also be explicitly added for incoming TCP connections or UDP traffic. When a packet arrives on the regular interface for the IP and port, it rewrites it using the recorded mapping and passes it to the VM. To the outside world, it simply appears that the Host is generating ordinary packets. To the VM, it appears as if it has a direct connection to the Local environment. For our pur-



ethx
Client
Client
LAN
"eth0"
ethy
VNET
Proxy
IP Network
ethz
VM
"eth0"
"Host Only" Network
vmnet0
VNET
Host
Ethernet Packet Captured by Promiscuous Packet Filter
Ethernet Packet Tunneled over TCP/SSL Connection
Ethernet Packet Injected Directly into VM interface

(a) Outbound traffic



ethx
Client
Client
LAN
"eth0"
ethy
VNET
Proxy
IP Network
ethz
VM
"eth0"
"Host Only" Network
vmnet0
VNET
Host
Ethernet Packet Injected Directly into Client LAN
Ethernet Packet Tunneled over TCP/SSL Connection
Ethernet Packet Captured by Promiscuous Packet Filter

(b) Inbound traffic

Figure 1. VNET configuration for a single remote virtual machine. Multiple virtual machines on the Host are possible, as are multiple hosts. Only a single Proxy is needed, and it can be the same as the Client. (a) Outbound traffic, (b) Inbound traffic.

poses, NAT networking is insufficient because it is painful to make incoming traffic work correctly as the mappings must be established manually. Furthermore, in some cases it would be necessary for the IP address of the virtual machine to change when it is migrated, making it impossible to maintain connections.

4.2 A bridge with long wires

In essence, VNET provides bridged networking, except that the VM is bridged to the Remote network, the network of the Client. VNET consists of a client and a server. The client is used simply to instruct servers to do work on its behalf. Each physical machine that can instantiate virtual machines (a Host) runs a single VNET server. At least one machine on the user's network also runs a VNET server. We refer to this machine as the Proxy. The user's machine is referred to as the Client. The Client and the Proxy can be the same machine. VNET consists of approximately 4000 lines of C++.

Figure 1 helps to illustrate the operation of VNET. VNET servers are run on the Host and the Proxy and are connected using a TCP connection that can optionally be encrypted using SSL. The VNET server running on the Host opens the Host's virtual interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose source address is that of the VM's virtual interface. The VNET server on the Proxy opens the Proxy's physical interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose destination address is that of the VM's virtual interface or is the Ethernet broadcast and/or (optionally) multicast addresses. To avoid loops, the packet must also not have a source address matching the VM's address. In each case, the VNET server is using the Berkeley packet filter interface [26] as implemented in libpcap, functionality available on all Unix platforms, as well as Microsoft Windows.

When the Proxy's VNET server sees a matching packet, it serializes it to the TCP connection to the Host's VNET server. On receiving the packet, the Proxy's VNET server directly injects the packet into the virtual network interface of the Host (using libnet, which is built on packet sockets, also available on both Unix and Windows) which causes it to be delivered to the VM's virtual network interface. Figure 1(a) illustrates the path of such outbound traffic. When the Host's VNET server sees a matching packet, it serializes it to the Proxy's VNET server. The Proxy's VNET server in turn directly injects it into the physical network interface card, which causes it to be sent on the LAN of the Client. Figure 1(b) illustrates the path of such inbound traffic.

The end-effect of such a VNET *Handler* is that the VM appears to be connected to the Remote Ethernet network exactly where the Proxy is connected. A Handler is identified by the following information:

- IP addresses of the Host and Proxy
- TCP ports on the Host and Proxy used by the VNET servers
- Ethernet devices used on the Host and Proxy
- Ethernet addresses which are proxied. These are typically the address of the VM and the broadcast address, but a single handler can support many addresses if needed.
- Roles assigned to the two machines (which is the Host and which is the Proxy)

A single VNET server can support an arbitrary number of handlers, and can act in either the Host or Proxy role for each. Each handler can support multiple addresses. Hence, for example, the single physical interface on a Proxy could provide connectivity for many VMs spread over many sites. Multiple Proxies or multiple interfaces in a single Proxy could be used to increase bandwidth, up to the limit of the User's site's bandwidth to the broader network.

Because VNET operates at the data link layer, it is agnostic about the network layer, meaning protocols other than IP can be used. Furthermore, because we keep the MAC address of the VM's virtual Ethernet adaptor and the LAN to which it appears to be connected fixed for the lifetime of the VM, migrating the VM does not require any participation from the VM's OS, and all connections remain open after a migration.

A VNET client wishing to establish a handler between two VNET servers can contact either one. This is convenient, because if only one of the VNET servers is behind a NAT firewall, it can initiate the handler with an outgoing

Command	Description
HELLO passwd version	Establish Session
DONE	Finish Session
DEVICES?	Return available network interfaces
HANDLERS?	Return currently running handlers
CLOSE handler	Tear down an existing handler
HANDLE remotepasswd	Establish a handler
local_config	(Described in text)
local_device	
remote_config	
remote_address	
remote_port	
remote_device	
macaddress+	
BEGIN local_config	Establish a handler
local_device	(Described in text)
remote_config	
remote_device	
macaddress+	

Figure 2. VNET interface.

connection through the firewall. If the client is on the same network as the firewall, VNET then requires only that a single port be open on the other site's firewall. If it is not, then both sites need to allow a single port through. If the desired port is not permitted through, there are two options. First, the VNET servers can be configured to use a common port. Second, if only SSH connections are possible, VNET's TCP connection can be tunneled through SSH.

4.3 Interface

VNET servers are run on the Host and the Proxy. A VNET client can contact any server to query status or to instruct it to perform an action on its behalf. The basic protocol is text-based, making it readily scriptable, and bootstraps to binary mode when a Handler is established. Optionally, it can be encrypted for security. Figure 2 illustrates the interface that a VNET Server presents.

Session establishment and teardown: The establishment of session with a VNET server is initiated by a VNET client or another server using the HELLO command. The client authenticates by presenting a password or by using an SSL certificate. Session teardown is initiated by the VNET client using the DONE command.

Handler establishment and teardown: After a VNET client has established a session with a VNET server, it can ask the server to establish a Handler with another server. This is accomplished using the HANDLE command. As shown in Figure 2, the arguments to this command are the parameters that define a Handler as described earlier. Here, `local_config` and `remote_config` refer to the Handler roles. In response to a HANDLE command, the server will establish a session with the other server in the Handler pair, authenticating as before. It will then issue a BEGIN command to inform the other VNET server of its intentions. If the other server agrees, both servers will bootstrap to a binary protocol for communicating Ethernet packets. The Handler will remain in place until one of the servers closes the TCP connection between them. This can be initiated by a client using the CLOSE command, directed at either server.

Status Enquiry: A client can discover a server's available network interfaces (DEVICES?) and what Handlers it is currently participating in (HANDLERS?).

4.4 Performance

Our goal for VNET was to make it easy to convey the network management problem induced by VMs to the home network of the user where it can be dealt with using familiar techniques. However, it is important that VNET's overhead not be prohibitively high, certainly not in the wide area. From the strongest to the weakest goal, VNET's performance should be

- in line with what the physical network is capable of,
- comparable to other networking solutions that don't address the network management problem, and
- sufficient for the applications and scenarios where it is used.

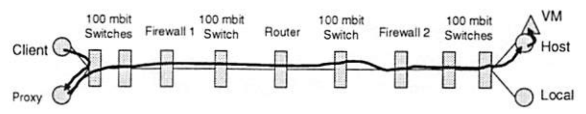
We have found that our implementation meets the later two goals, and, in many cases, meets the first, strongest goal as well.

Metrics

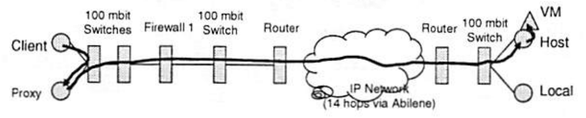
Latency and throughput are the most fundamental measures used to evaluate the performance of networks. The time for a small transfer is dominated by latency, while that for a large transfer is dominated by throughput. Interactivity, which is often dominated by small transfers, suffers if latencies are either high or highly variable [8]. Bulk transfers suffer if throughput is low. Our measurements were conducted on working days (Monday through Thursday) in the early morning to eliminate time-of-day effects.

Latency: To measure latency, we used the round-trip delay of an ICMP echo request/response pair (i.e., ping), taking samples over hour-long intervals. We computed the average, minimum, maximum and standard deviation of these measurements. Here, we report the average and standard deviation. Notice that this measure of latency is symmetric.

Throughput: To measure average throughput, we use the *ttcp* program. Ttcp is commonly used to test TCP and UDP performance in IP networks. Ttcp times the transmission and reception of data between two systems. We use a socket buffer size of 64 KBytes and transfer a total of 1



(a) Local Area Configuration



(b) Wide Area Configuration

Figure 3. VNET test configurations for the local area (a) and the wide area (b). Local area is between two labs in the Northwestern CS Department. Wide Area is between the first of those labs and a lab at Carnegie Mellon.

GB of data in each test. VNET's TCP connection also uses a socket buffer size of 64 KBytes. TCP socket buffer size can limit performance if it is less than the bandwidth-delay product of the network path, hence our larger-than-default buffers. All throughput measurements were performed in both directions.

Testbeds

Although VNET is targeted primarily for wide-area distributed computing, we evaluated performance in both a LAN and a WAN. Because our LAN testbed provides much lower latency and much higher throughput than our WAN testbed, it allows us to see the overheads due to VNET more clearly. The Client, Proxy, and Host machines are 1 GHz Pentium III machines with Intel Pro/100 adaptors. The virtual machine uses VMware GSX Server 2.5, with 256 MB of memory, 2 GB virtual disk and RedHat 7.3. The network driver used is vmxnet.

Our testbeds are illustrated in Figure 3. The LAN and WAN testbeds are identical up to and including the first router out from the Client. This portion is our firewalled lab in the Northwestern CS department. The LAN testbed then connects, via a router which is under university IT control (not ours), to another firewalled lab in our department which is a separate, private IP network. The WAN testbed instead connects via the same router to the Northwestern backbone, the Abiline network, the Pittsburgh Supercomputing Center, and two administrative levels of the campus network at Carnegie Mellon, and finally to an lab machine there. Notice that even a LAN environment can exhibit the network management problem. It is important to stress that the only requirement that VNET places on either of these complex environments is the ability to create a TCP connection between the Host and Proxy in some way.

We measured the latency and throughput of the underlying "physical" IP network, VMWare's virtual networking options, VNET, and of SSH connections:

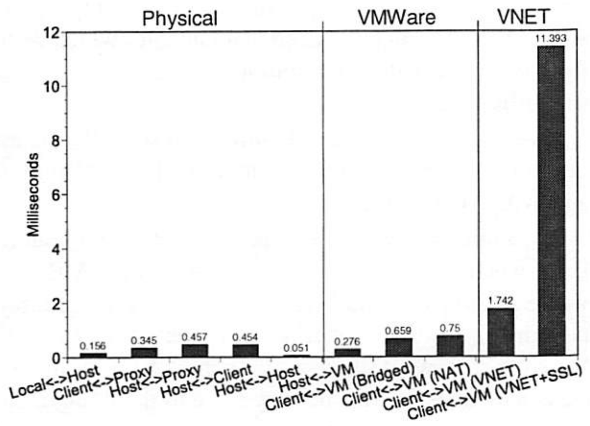
- *Physical:* VNET transfers Ethernet packets over multiple hops in the underlying network. We measure equivalent hops, and also end-to-end transfers, excepting the VM.
    - $Local \leftrightarrow Host$: Machine on the Host's LAN to/from the Host.
    - $Client \leftrightarrow Proxy$: Analogous to the first hop for an outgoing packet in VNET and the last hop for an incoming packet.
    - $Host \leftrightarrow Proxy$: Analogous to the TCP connection of a Handler, the tunnel between the two VNET servers.
    - $Host \leftrightarrow Client$: End-to-end except for the VM.
    - $Host \leftrightarrow Host$: Internal transfer on the Host.
- *VMWare:* Here we consider the performance of all three of VMWare's options, described earlier.
    - $Host \leftrightarrow VM$: Host-only networking, which VNET builds upon.
    - $Client \leftrightarrow VM$ *(Bridged)*: Bridged networking. This leaves the network administration problem at the remote site.
    - $Client \leftrightarrow VM$ *(NAT)*: NAT-based networking. This partially solves the network administration problem at the remote site at the layer 3, but creates an asymmetry between incoming and outgoing connections, and does not support VM migration. It's close to VNET in that network traffic is routed through a user-level server.
- *VNET:* Here we use VNET to project the VM onto the Client's network.
    - $Client \leftrightarrow VM$ *(VNET)*: VNET without SSL
    - $Client \leftrightarrow VM$ *(VNET+SSL)*: VNET with SSL
- *SSH:* Here we look at the throughput of an SSH connection between the Client and the Host to compare with VNET with SSL.
    - $Host \leftrightarrow Client$ *(SSH)*

Discussion

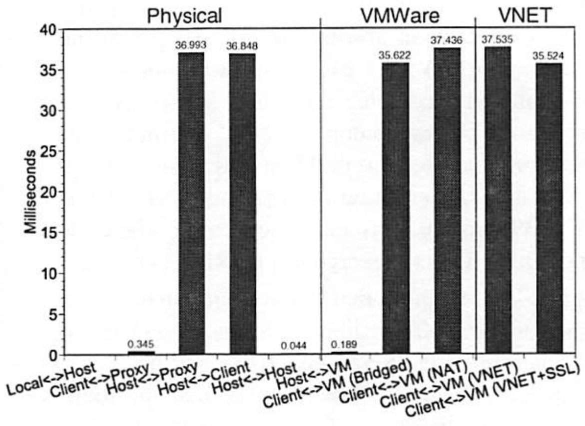
The results of our performance tests are presented in Figures 4 through 6.

Average latency: Figure 4 shows the average latency in the LAN (Figure 4(a)) and WAN (Figure 4(b)).

In Figure 4(a), we see that the average latency on the LAN when using VNET without SSL is 1.742 ms. It is important to understand exactly what is happening. The Client is sending an ICMP echo request to the VM. The request is first intercepted by the Proxy, then sent to the Host, and finally the Host sends it to the VM (see Figure 3(a)). The reverse path for the echo reply is similar. These three distinct pieces have average latencies of 0.345 ms, 0.457 ms, and 0.276 ms, respectively, on the physical network, which

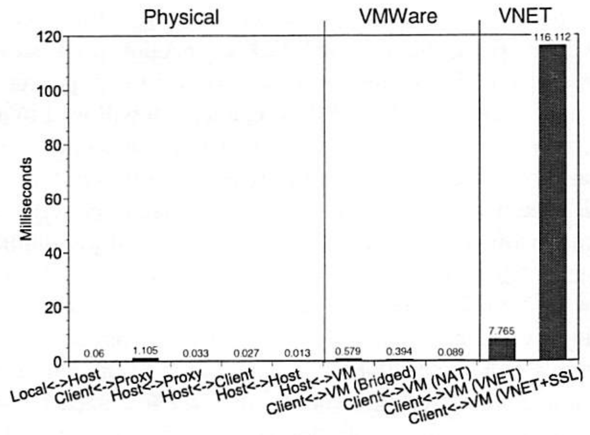


(a) Local Area

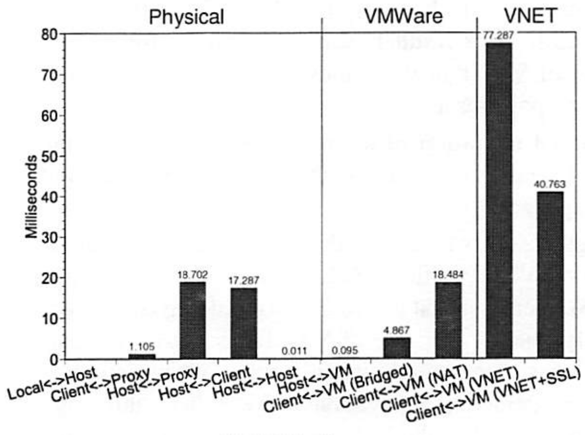


(b) Wide Area

Figure 4. Average latency

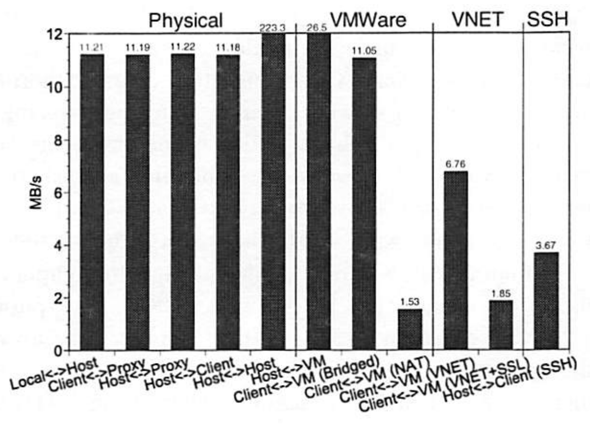


(a) Local Area

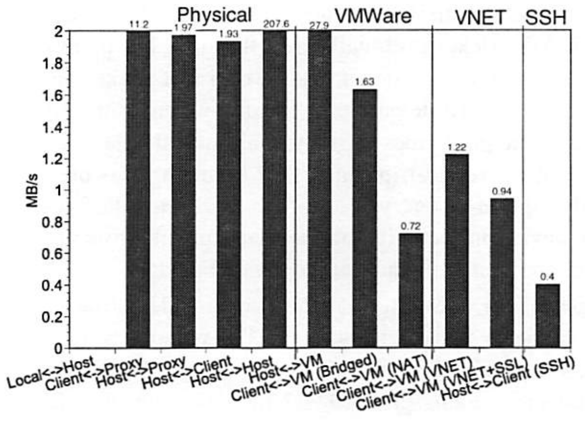


(b) Wide Area

Figure 5. Standard deviation of latency



(a) Local Area



(b) Wide Area

Figure 6. Bandwidth

totals 1.078 ms. In the LAN, VNET without SSL increases latency by 0.664 ms, or about 60%. We claim that this is not prohibitive, especially in absolute terms. Hence we note that the operation of VNET over LAN does not add prohibitively to the physical latencies. The VMWare NAT option, which is the closest analog to VNET, except for moving the network management problem, has about 1/2 of the latency. When SSL encryption is turned on, VNET latency grows to 11.393 ms, 10.3 ms and a factor of 10 higher than what is possible on the (unencrypted) physical network.

In Figure 4(b), we note that the average latency on the WAN when using VNET without SSL is 37.535 ms and with SSL encryption is 35.524 ms. If we add up the constituent latencies as done above, we see that the total is 37.527 ms. In other words, VNET with or without SSL has average latency comparable to what is possible on the physical network in the WAN. The average latencies seen by VMWare's networking options are also roughly the same. In the wide area, average latency is dominated by the distance, and we get the benefits of VNET with negligible additional cost. This result is very encouraging for the deployment of VNET in the context of grid computing, our primary purpose for it.

Standard deviation of latency: Figure 5 presents the standard deviation of latency in the LAN (Figure 5(a)) and WAN (Figure 5(b)).

In Figure 5(a), we see that the standard deviation of latency using VNET without SSL in the LAN is 7.765 ms, while SSL increases that to 116.112 ms. Adding constituent parts only totals 1.717 ms, so VNET has clearly dramatically increased the variability in latency, which is unfortunate for interactive applications. We believe this large variability is because the TCP connection between VNET servers inherently trades packet loss for increased delay. For the physical network, we noticed end-to-end packet loss of approximately 1%. VNET packet losses were nil. VNET resends any TCP segment that contains an Ethernet packet that in turn contains an ICMP request/response. This means that the ICMP packet eventually gets through, but is now counted as a high delay packet instead of a lost packet, increasing the standard deviation of latency we measure. A histogram of the ping times shows that almost all delays are a multiple of the round-trip time. TCP tunneling was used to have the option of encrypted traffic. UDP tunneling reduces the deviation seen, illustrating that it results from our specific implementation and not the general design.

In Figure 5(b), we note that the standard deviation of latency on the WAN when using VNET without SSL is 77.287 ms and with SSL is 40.783 ms. Adding the constituent latencies totals only 19.902 ms, showing that we have an unexpected overhead factor of 2 to 4. We again suspect high packet loss rates in the underlying network lead to retransmissions in VNET and hence lower packet loss rates, but a higher standard deviation of latency. We measured a 7% packet loss rate in the physical network compared to 0% with VNET. We again noticed that latencies which deviated from the average did so in multiples of the average latency, supporting our explanation.

Average Throughput: Figure 6 presents the measurements for the average throughput in the LAN (Figure 6(a)) and WAN (Figure 6(b)).

In Figure 6(a), we see that the average throughput in the LAN when using VNET without SSL is 6.76 MB/sec and with SSL drops to 1.85 MB/sec, while the average throughput for the physical network equivalent is 11.18 MB/sec. We were somewhat surprised with the VNET numbers. We expected that we would be very close to the throughput obtained in the physical network, similar to those achieved by VMWare's host-only and bridged networking options. Instead, our performance is lower than these, but considerably higher than VMWare's NAT option.

In the throughput tests, we essentially have one TCP connection (that used by the ttcps running on the VM and Client) riding on a second TCP connection (that between the two VNET servers on Host and Proxy). A packet loss in the underlying VNET TCP connection will lead to a retransmission and delay for the ttcp TCP connection, which in turn could time out and retransmit itself. On the physical network there is only ttcp's TCP. Here, packet losses might often be detected by the receipt of triple duplicate acknowledgements followed by fast retransmit. However, with VNET, more often than not a loss in the underlying TCP connection will lead to a packet loss detection in ttcp's TCP connection by the expiration of the retransmission timer. The difference is that when a packet loss is detected by timer expiration the TCP connection will enter slow start, dramatically slowing the rate. In contrast, a triple duplicate acknowledgement does not have the effect of triggering slow start.

In essence, VNET is tricking ttcp's TCP connection into thinking that the round-trip time is highly variable when what is really occurring is hidden packet losses. In general, we suspect that TCP's congestion control algorithms are responsible for slowing down the rate and reducing the average throughput. This situation is somewhat similar to that of a *split TCP connection*. A detailed analysis of the throughput in such a case can be found elsewhere [34]. The use of encryption with SSL further reduces the throughput.

In Figure 6(b), we note that the average throughput over the WAN when using VNET without SSL encryption is 1.22 MB/sec and with SSL is 0.94 MB/sec. The average throughput on the physical network is 1.93 MB/sec. Further, we note that the throughput when using VMWare's bridged networking option is only slightly higher than the case where VNET is used (1.63 MB/sec vs. 1.22 MB/sec), while VMWare NAT is considerably slower. Again, as described above, this difference in throughput is probably due to the overlaying of two TCP connections. Notice, however, that the difference is much less than that in the LAN as now there are many more packet losses that in both cases will be detected by ttcp's TCP connection by the expiration of the retransmission timer. Again, the use of encryption with SSL further reduces the throughput.

We initially thought that our highly variable latencies (and corresponding lower-than-ideal TCP throughput) in VNET were due to the priority of the VNET server processes. Conceivably, the VNET server could respond slowly if there were other higher or similar priority processes on the Host, Proxy, or both. To test this hypothesis we tried giving the VNET server processes maximum priority, but this did not change delays or throughput. Hence, this hypothesis was incorrect.

We also compared our implementation of encryption using SSL in the VNET server to SSH's implementation of SSL encryption. We used SCP to copy 1 GB of data from the Host to the Client in both the LAN and the WAN. SCP uses SSH for data transfer, and uses the same authentication and provides the same security as SSH. In the LAN case we found the SCP transfer rate to be 3.67 MB/sec compared to the 1.85 MB/sec with VNET along with SSL encryption. This is an indication that our SSL encryption implementation overhead is not unreasonable. In the WAN the SCP transfer rate was 0.4 MB/sec compared to 0.94 MB/sec with VNET with SSL. This further strengthens the claim that our implementation of encryption in the VNET server is reasonably efficient.

Comparing with VMWare NAT: The throughput obtained when using VMWare's NAT option was 1.53 MB/sec in the LAN and 0.72 MB/sec in the WAN. This is significantly lower than the throughput VNET attains both in the LAN and WAN (6.76 MB/sec and 1.22 MB/sec, respectively). As described previously in Section 4.1, VMWare's NAT is a user-level process, similar in principle to a VNET server process. That VNET's performance exceeds that of VMWare NAT, the closest analog in VMWare to VNET's functionality, is very encouraging.

Summary: The following are the main points to take away from our performance evaluation:

- Beyond the physical network and the VMWare networking options, VNET gives us the ability to shift the network management problem to the home network of the client.
- The extra average latency when using VNET deployed over the LAN is quite low while the overhead over the WAN is negligible.
- VNET has considerably higher variability in latency than the physical network. This because it automatically retransmits lost packets. If the underlying network has a high loss rate, then this will be reflected as higher latency variability in VNET. Hence, using VNET, in its current implementation, produces a trade: higher variability in latency for zero visible packet loss.
- VNET's average throughput is lower than that achievable in the underlying network, although not dramatically so. This appears to be due to an interaction between two levels of TCP connections. We are working to fix this.
- VNET's average throughput is significantly better than that of the closest analog, both in terms of functionality and implementation, in VMWare, NAT.
- Using VNET encryption increases average latency and standard deviation of latency by a factor of about 10 compared to the physical network. Encryption also decreases throughput. The VNET encryption results are comparable or faster than those using SSH.

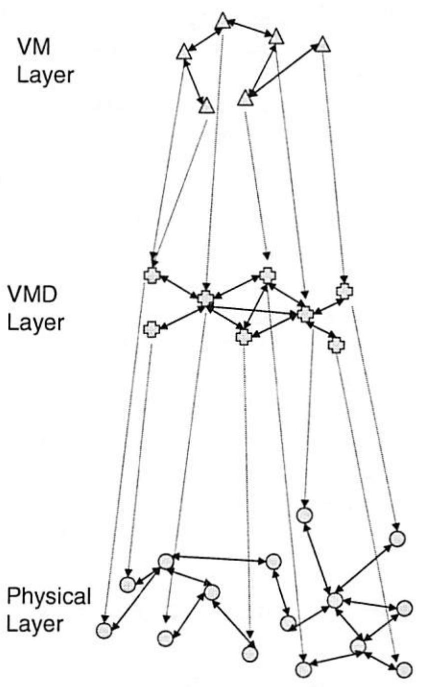
We find that the overheads of VNET, especially in the WAN, are acceptable given what it does, and we are working to make them better. Using VNET, we can transport the network management problem induced by VMs back to the home network of the user, where it can be readily solved, and we can do so with acceptable performance.

5 Towards an adaptive overlay

We designed and implemented VNET in response to the network management problems encountered when running VMs at (potentially multiple) sites where the user has no administrative connection. However, we have come to believe strongly that the overlays like it, specifically designed to support virtual machine computing, have great potential as the mechanisms for adaptation and for exploiting the special features of some networks.

An overlay network has an ideal vantage point to monitor the underlying physical network and the applications running the VMs. Using this information, it can adapt to the communication and computation behavior of the VMs, changing its topology and routing rules, and moving VMs. Requiring code modifications, extensions, or the use of particular application frameworks has resulted in limited adoption of adaptive application technologies. Here, adaptation could be retrofitted with *no modifications* to the operating system and applications running in the virtual machine, and could be completely transparent to the running VMs. Similarly, if the overlay is running on a network that can provide extended services, such as reservations or light-path setup and teardown in an optical network, it could use these features on behalf of an unmodified operating system and its applications.

We are now designing a second generation VNET implementation that will support this vision. The second generation VNET will include support for arbitrary topologies and routing, network and VM monitoring, and interfaces for adaptative control of the overlay, including VM migration, and for using underlying resource reservation mechanisms. In the following, we describe these extensions and then elaborate on the adaptation problem.



G_{vm}(VM,VE)	virtual machine graph (directed)
vm_i in VM	virtual machine (VM) (node)
$ve_{i,j}$ in VE	exists if vm_i sends data to vm_j (edge)
fixed(vm_i,vd_k)	true if vm_i may only be assigned to vd_k
comp(vm_i)	compute rate demand of vm_i
size(vm_i)	size of vm_i in bytes
bw($ve_{i,j}$)	bandwidth demand on $ve_{i,j}$
lat($ve_{i,j}$)	latency demand on $ve_{i,j}$

vmap(vm_i)	***assignment of vm_i to appropriate vd***
	many-to-one, dynamic

G_{vmd}(VD,VDE)	VM daemon graph (directed)
	VM daemons host zero or more VMs
vd_i in VD	VM Daemon (VMD) (node)
$vde_{i,j}$ in VDE	***exists if vd_i sends data directly to vd_j (edge)***
	dynamic
route(vd_i,vd_j)	***path (<vd_k>) taken by data sent from vd_i to vd_j***
route($ve_{i,j}$)	***dynamic***
mcomp(vd_i)	measured compute rate at vd_i
mbw($vde_{i,j}$)	measured available bandwidth on $vde_{i,j}$
mlat($vde_{i,j}$)	measured latency on $vde_{i,j}$

vdmap(vd_i)	***assignment of vd_i to appropriate host***
	static
route($vde_{i,j}$)	***route that $vde_{i,j}$ takes in network topology graph***

G_{net} (N,E)	Network topology graph (directed)
	VM Daemons run only on hosts
n_i in N	router or host
host(n_i)	true if node is a host
$e_{i,j}$ in E	exists if n_i sends data to n_j (edge)
route(n_i,n_j)	path (<n_k>) taken by data sent from vd_i to vd_j
mbw(n_i)	backplane bandwidth of a router or network
mbw($e_{i,j}$)	measured available bandwidth on edge
	may not exist
mlat($e_{i,j}$)	measured latency on edge
	may not exist

Figure 7. Terminology.

5.1 Terminology

Figure 7 serves as a point of reference for the terminology we use in describing our design and the problems it induces. On the left hand side, we can see the three layers: the virtual machines themselves (VM layer), the virtual machine daemons that host them (VMD layer), and the physical network resources on which the VMDs run (Physical layer). The graph of VM Daemons is the overlay itself. A VM daemon or VMD is a generalization of a VNET server that is able to manage VMs and measure their traffic and the characteristics of the underlying network. The nodes and edges at the VM layer are mapped to nodes and routes at the VMD layer. The physical layer is the underlying IP network itself. The nodes of the VMD layer are mapped to the end-systems of this network. The right hand side of Figure 7 shows the symbols we use at each layer and the mapping between layers. The boldfaced symbols represent where adaptation and the use of underlying resource mechanisms can take place.

The VM layer consists of the individual VMs ($vm_i \in VM$) and the communication edges among them ($ve_{i,j} \in VE$), represented as a graph ($G_{vm}(VM, VE)$). If vm_i sends to vm_j, then there is an edge $ve_{i,j}$ in VE. The VM layer is essentially a representation of the demands that the user's VMs are placing on the system. $bw(ve_{i,j})$ and $lat(ve_{i,j})$ are the bandwidth and latency requirements of communication between vm_i and vm_j. $comp(vm_i)$ is the computational demand of vm_i, while $size(vm_i)$ is the total size of its machine image.

The VMD layer consists of VM daemons ($vd_i \in VD$), and the communication edges among them ($vde_{i,j} \in VDE$), represented as a graph ($G_{vmd}(VD, VDE)$). If vd_i sends to vd_j, there must be a route ($route(vd_i, vd_j)$) between them. A virtual machine vm_i is assigned to a single VMD ($vmap(vm_i)$). Hence, an edge $ve_{i,j}$ in the VM layer corresponds to a route $route(ve_{i,j}) = route(vmap(vm_i), vmap(vm_j))$ in the VMD layer. Multiple VMs may be assigned to a single VMD. At the VMD layer, we also maintain the measured bandwidth and latency of edges in the G_{vmd}, $mbw(vde_{i,j})$, $mlat(vde_{i,j})$, and the computational rate at each node ($mcomp(vd_i)$).

The physical layer consists of the underlying topology, $G_{net}(V, E)$, where the nodes $v_i \in V$ are the routers and hosts at the IP layer and $e_{i,j} \in E$ are the links. $route(v_i, v_j)$ are the routes chosen by the network, and $host(v_i)$ is true if v_i is a host. In addition, we may be able to measure the bandwidth and latency of the elements of a link ($mbw(e_{i,j})$, $mlat(e_{i,j})$), the backplane bandwidth of a router ($mbw(v_i)$), and the raw computational power of a host ($mcomp(v_i)$). Each VMD is assigned to a single host in the physical layer, and each host has at most a single VMD. This mapping is via $vdmap(vd_i)$. Each edge in the G_{vmd} turns into a route $route(vde_{i,j})$ at the physical layer.

5.2 Supporting arbitrary topologies and routing

Currently, a VMD (VNET server) tunnels Ethernet traffic for a particular address to another VMD over a TCP connection, a relationship we refer to as a handler, as described in Section 4.2. In principle, multiple handlers can be multiplexed over a single TCP connection. Hence, the edges in the VMD graph are simply TCP connections. In such a configuration, the VMDs supporting a user form an overlay network with a star topology centered on the proxy machine on the user's network. All messages are routed through the proxy machine.

It is straightforward to see why it is possible to support arbitrary topologies. Each VMD can effectively behave like a switch or router, sending an packet that arrives on a TCP connection to another connection instead of injecting it onto the local network. Indeed, for Ethernet topologies, we can simply emulate the Ethernet switch protocols, as in VIOLIN [22]. Each Ethernet packet contains a source and destination address. A modern Ethernet switch learns the location of Ethernet addresses on its ports based on the source addresses of traffic it sees. Switches also run a distributed algorithm that assures that they form a spanning tree topology.

Because VNET understands that it is supporting VMs, it can go beyond simply emulating an IP or Ethernet network, however. For example, hierarchical routing of Ethernet packets is possible because the Ethernet addresses of the user's VMs are not chosen by Ethernet card vendors, but are assigned by VNET.

5.3 Monitoring the VMs and the network

The VMD layer is ideally placed to monitor both the resource demands placed by the VMs and the resource supplies offered by the underlying physical network, with minimal active participation from either.

Each VMD sees every incoming/outgoing packet to/from each of the VMs that it is hosting. Given a matrix representation of G_{vm}, if the VMD is hosting vm_i it knows the ith row and ith column. Collectively, the VMDs know all of G_{vm}, so a reduction could be used to give each one a full copy of G_{vm}. Hence, without modifying the OS or any operating systems, the VMDs can recover the application topology.

Each VMD is also in a good position to infer the bandwidth and latency demands for each edge, $bw(ve_{i,j})$, $lat(ve_{i,j})$, the computational demand of each VM, $comp(vm_i)$, and the total size of each VM image, $size(vm_i)$ corresponding to the VMs that it is hosting. Again, a reduction would give each VMD a global picture of the the resource demands. Beyond inference, this information could also be provided directly by a developer or administrator without any modifications to the applications or operating system.

VMDs transfer packets on behalf of the VMs they host. An outgoing packet from a VM is routed through it's hosting VMD, then through zero or more transit VMDs, the host VMD of the destination VM, and finally to the destination VM. When such a message is transfered from vd_i to vd_j, the transfer time is a free measurement of the corresponding path $route(vde_{i,j})$ in the underlying network. From collections of such measurements, the two VMDs can derive $mbw(vde_{i,j})$ and $mlat(vde_{i,j})$ using known techniques [5]. A VMD can also periodically measure the available compute rate of its host ($mcomp(vd_i)$) using known techniques [6, 39].

Network monitoring tools such as Remos [5] and NWS [40] can, in some cases, determine the physical layer topology and measure its links, paths, and hosts.

5.4 VM assignment problems

Let us define some additional terminology. Each VM computes at some actual rate:

$$ComputeRate(vm_i)$$

Each pair of VMs have some actual bandwidth and latency:

$$PathBW(vm_i, vm_j) = PathBW(ve_{i,j}) = \min_{vde \in route(vmap(vm_i), vmap(vm_j))} mbw(vde)$$

$$PathLatency(vm_i, vm_j) = PathLatency(ve_{i,j}) = \sum_{vde \in route(vmap(vm_i), vmap(vm_j))} mlat(vde)$$

A VMD has an allocated compute rate, which is the sum of all the rates of VMs mapped to it:

$$AllocatedComputeRate(vd_i) = \sum_{vm \in VM : vmap(vm) = vd_i} comp(vm)$$

Similarly, an edge between two VMDs has an allocated bandwidth:

$$AllocatedBandwidth(vde_{i,j}) = \sum_{ve \in VE : vde_{i,j} \in route(ve)} bw(ve)$$

The VM assignment problem is to find a *vmap* function that meets the following requirements:

- Complete: $\forall vm \in VM : vmap(vm)$ exists.
- Compliant: $fixed(vm_i, vd_j) \Rightarrow vmap(vm_i) = vd_j$

- Computationally feasible: $\forall vm \in VM : ComputeRate(vm) \geq comp(vm)$ and $\forall vd \in VD : AllocatedComputeRate(vd) \leq mcomp(vd)$
- Communication feasible: $\forall ve \in VE :$ $PathLatency(ve) \leq lat(ve) \wedge PathBW(ve) \geq bw(ve)$ and $\forall vde \in VDE : AllocatedBW(vde) \leq mbw(vde)$
- Routable: $\forall ve \in VE : route(ve)$ exists.

We define four variants of the VM assignment problem. The first two are offline versions while the second two are online problems.

Simple offline VM assignment problem: Given G_{vm} and its associated functions $fixed$, $comp$, $size$, bw, and lat, and G_{vmd} and its associated functions $route$, $mcomp$, mbw, and $mlat$, choose a $vmap$ that meets the $vmap$ requirements. The VMDs are fixed, as are their overlay topology and routing rules. We envision three situations in which this problem arises. The first is if the user has a private VMD network and runs a multi-VM application on it. The problem needs to be solved at program startup, and thereafter whenever the communication patterns have changed dramatically. The second case is when multiple users can map their own virtual machines to a shared VMD infrastructure. In this situation, the VMs of other users can be treated as $fixed$. The problem also occurs if it is the VMD infrastructure that determines the mapping. In that situation, the problem can be solved with no VMs $fixed$ whenever a new set of VMs enters or leaves the system, or periodically.

Complex offline VM assignment problem: Given G_{vm} and its associated functions, determine $vmap$, VDE, and $route$ such that $vmap$ meets the $vmap$ requirements. Here, the VMDs are fixed, but their overlay topology and its routing rules can be chosen to help find a suitable $vmap$. We see this problem occurring in two contexts. The first is if the user has a private VMD network that supports topology and routing changes. The second is for a shared VMD overlay where the VMDs solve the problem collectively over all running VMs.

Simple online VM assignment problem: Given an existing G_{vm}, G_{vmd}, and their associated functions, an existing $vmap$, and a new G'_{vm}, determine $vmap' = f(vmap, G_{vm}, G'_{vm})$ such that it meets the $vmap$ requirements.

Complex online VM assignment problem: Given an existing G_{vm}, G_{vmd}, and their associated functions, an existing $vmap$, $route$, and a new G'_{vm}, determine a new $(vmap', VDE', route') = f(vmap, G_{vm}, G'_{vm})$ such that $vmap$ meets the requirements.

5.5 Connected components

The simple online VM assignment problem can be formulated in terms of connected components. The connected components of a graph are a partitioning of the graph into subgraphs. A connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, there is a path connecting them. Hence two vertices are in the same connected component if and only if there exists a path between the vertices. If a graph contains only one connected component then it is said to be a connected graph.

The directed graph G_{vm} is a connected graph while the directed graph G_{vmd} may or may not be connected. It will be connected if and only if $\forall\ vd_i \in VD$ there $\exists$ at least one $vm_i \in VM$ such that $vmap(vm_i) = vd_i$. Note that since G_{vm} is connected, the graph of $vd \in VD$ where $\forall\ vd$ there $\exists\ vmap$ such that $vd = vmap(vm_i)$ such that $vm_i \in VM$, will also be connected.

Given an existing G_{vm}, G_{vmd}, and their associated functions, we calculate $cost(vde_{i,j})$ where

$$cost(vde_{i,j}) = f(mbw(vde_{i,j}), mlat(vde_{i,j}))\forall vde_{i,j} \in VL$$

$cost(vde_{i,j})$ is a measure of the network *cost value* associated with each edge in the VM daemon graph G_{vmd}. The cost value would be an integrated metric, a function of the measured available bandwidth, $mbw(vde_{i,j})$, and latency, $lat(vde_{i,j})$, on the edges in the VM daemon graph G_{vmd}.

The VM assignment problem may now be described as:

Input:

- G_{vm}, G_{vmd}, and their associated functions
- an existing $vmap$
- a connected component $VDC \subseteq VD$ such that $\forall vd_i \in VDC$ there $\exists$ at least one $vm_i \in VM$ such that $vmap(vm_i) = vd_i$
- a cost value $cost(vde_{i,j})\ \forall vde_{i,j} \in VDE$

Output:

- a connected component $VDC' \subseteq VD$ such that $\forall vd_i \in VDC'$ there $\exists$ at least one $vm_i \in VM$ such that $vmap'(vm_i) = vd_i$ and $\sum_{\forall vd_i, vd_j \in VDC'} cost(vde_{i,j})$ is minimum over the set of all possible connected components

Algorithm:

- Each connected component (VDC, VDC', VDC'', etc) represents one particular assignment of $vm_i \in VM$ to $vd_i \in VD$, i.e. a particular $vmap$ meeting all the $vmap$ requirements
- So from amongst all such possible connected components (assignments) we choose that which has the least cost associated with it, i.e. $\sum_{\forall vd_i, vd_j \in VDC'} cost(vde_{i,j})$ is minimum over the set of all possible connected components
- This is equivalent to enumerating all the possible connected components of size | VM | and then choosing the one which has the least cost associated with it.

5.6 Engineering a VMD overlay

For a VMD infrastructure that supports a single user, it is sensible to think of solving the VM assignment problems by exploiting simultaneously the freedom to change the VMD topology and routes, as well as to move the VMs. However, if the VMD infrastructure is to be shared among multiple users, a two stage approach, engineering a sensible general-purpose VMD topology and then doing dynamic assignment of VMs to that topology, is likely to be more sensible.

One approach is to require that the topology of the VMDs, G_{vmd} be a mesh or a hypercube. This induces a network engineering problem, that of finding $vdmap$ such that the chosen topology behaves close to our expectations for its type in terms of the bandwidth and latency of the edges. For both meshes and hypercubes, this means that each edge should be equal in terms of bandwidth and latency. Given that the underlying overlay has such as simple, regular topology, we would assign groups of VMs that exhibit a particular topology to partitions of the VMD graph. For example, it is well known that trees and neighbor patterns can be embedded in hypercubes [24]. If a parallel application running across a set of VMs exhibits one of these patterns, its VMs can be readily (and quickly) assigned to VMDs.

5.7 Exploiting resource reservations

Some networks support reservations of bandwidth along paths. For example, various optical networks support light-path set up, essentially allowing for an arbitrary topology to be configured. The ODIN software [25] provides an application with an API to exploit such networks. However, it is the onerous task of the programmer to determine the appropriate topology and then use the API to configure it. Here, the VMDs could do the same on behalf of the unmodified application, since they collectively know G_{vm}.

Light-path services, as well as traditionally ISP-level reservations such as DiffServ [28], could also be used to implement an engineered overlay that is shared by multiple users, as described previously.

6 Conclusions

A strong case can be made for grid computing using virtual machine monitors or virtual servers. In either case, the combination of a virtual machine's need for a network presence and the multi-site, multi-security-domain nature inherent in grid computing creates a challenging distributed network management problem. We have described and evaluated a tool, VNET, that addresses this problem by converting it into the familiar single-site network management problem. The combination of VNET and similar virtual network tools and virtual machines present an opportunity: adaptation and exploitation of resource reservations for existing, unmodified operating systems and applications. We described this opportunity in depth, pointing out specific adaptation problems and interactions with a specific resource reservation system. We are currently extending VNET to take advantage of this opportunity. VNET is publicly available and can be downloaded from http://plab.cs.northwestern.edu/Virtuoso.

Acknowledgements

We would like to thank David O'Hallaron, Julio Lopez, and Renato Figueiredo for giving us access to the machines we used in our wide area performance evaluation, and Ashish Gupta, Sonia Fahmy, and Dongyan Xu for helpful discussions on the overlay aspects of VNET.

References

[1] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (2001).

[2] BOLLINGER, S., AND MIDKIFF, S. Heuristic techniques for processor and link assignment in multicomputers. *IEEE Transactions on Computers 40*, 3 (March 1991).

[3] CAMPBELL, A., KOUNAVIS, M., VILLELA, D., VICENTE, J., MEER, H. D., MIKI, K., AND KALAICHELVAN, K. Spawning networks. *IEEE Network* (July/August 1999), 16–29.

[4] DIKE, J. A user-mode port of the linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference* (Atlanta, GA, October 2000).

[5] DINDA, P., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND MILLER, N. The architecture of the remos system. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)* (August 2001), pp. 252–265.

[6] DINDA, P. A. Online prediction of the running time of tasks. *Cluster Computing 5*, 3 (2002). Earlier version appears in HPDC 2001. Summary in SIGMETRICS 2001.

[7] EGEVANG, K., AND FRANCIS, P. The ip network address translator (nat). Tech. Rep. RFC 1631, Internet Engineering Task Force, May 1994.

[8] EMBLEY, D. W., AND NAGY, G. Behavioral aspects of text editors. *ACM Computing Surveys 13*, 1 (January 1981), 33–70.

[9] ENSIM CORPORATION. http://www.ensim.com.

[10] FERGUSON, P., AND HUSTON, G. What is a vpn? Tech. rep., Cisco Systems, March 1998.

[11] FIGUEIREDO, R., DINDA, P. A., AND FORTES, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd IEEE Conference on Distributed Computing (ICDCS 2003* (May 2003), pp. 550–559.

[12] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications 15*, 3 (2001).

[13] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[14] GLEESON, B., LIN, A., HEINANEN, J., ARMITAGE, G., AND MALIS, A. A framework for ip-based virtual private networks. Tech. Rep. RFC 2764, Internet Engineering Taskforce, February 2000.

[15] HAND, S., HARRIS, T., KOTSOVINOS, E., AND PRATT, I. Controlling the xenoserver open platform. In *Proceedings of OPENARCH 2003* (April 2003).

[16] HUA CHU, Y., RAO, S., SHESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001* (2001).

[17] IBM CORPORATION. White paper: S/390 virtual image facility for linux, guide and reference. GC24-5930-03, Feb 2001.

[18] IEEE 802.1Q WORKING GROUP. 802.1q: Virtual lans. Tech. rep., IEEE, 2001.

[19] ITALIANO, G., RASTOGI, R., AND YENER, B. Restoration algorithms for virtual private networks in the hose model. In *Procedings of IEEE INFOCOM 2002* (June 2002).

[20] JANNOTTI, J., GIFFORD, D., JOHNSON, K., KAASHOEK, M., AND JR., J. O. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI 2000* (October 2000).

[21] JIANG, X., AND XU, D. Soda: A service-on-demand architecture for application service hosting platforms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 2003)* (June 2003), pp. 174–183.

[22] JIANG, X., AND XU, D. Violin: Virtual internetworking on overlay infrastructure. Tech. Rep. CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[23] KWONG, K., AND ISHFAQ, A. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing 59*, 3 (1999), 381–422.

[24] LEIGHTON, T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[25] MAMBRETTI, J., WEINBERGER, J., CHEN, J., BACON, E., YEH, F., LILLETHUN, D., GROSSMAN, B., GU, Y., AND MAZZUCO, M. The photonic terastream: Enabling next generation applications through intelligent optical networking at igrid2002. *Future Generation Computer Systems 19*, 6 (August 2003), 897–908.

[26] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Prodeedings of USENIX 1993* (1993), pp. 259–270.

[27] MICROSOFT CORPORATION. Virtual server beta release.

[28] NICHOLS, K., BLAKE, S., BAKER, F., AND BLACK, D. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. Tech. rep., Internet Engineering Task Force, 1998.

[29] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (January/February 1998).

[30] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA 2003)* (October 2003).

[31] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM:, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)* (December 2002).

[32] SHI, S., AND TURNER, J. Routing in Overlay Multicast Networks. In *Proceedings of IEEE INFOCOM 2002* (June 2002).

[33] SUGERMAN, J., VENKITACHALAN, G., AND LIM, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[34] SUNDARARAJ, A. I., AND DUCHAMP, D. Analytical characterization of the throughput of a split tcp connection. Tech. rep., Department of Computer Science, Stevens Institute of Technology, 2003.

[35] TOWNSLEY, W., VALENCIA, A., RUBENS, A., PALL, G., ZORN, G., AND PALTER, B. Layer two tunneling protocol "l2tp". Tech. Rep. RFC 2661, Internet Engineering Task Force, August 1999.

[36] VIRTUOZZO CORPORATION. http://www.swsoft.com.

[37] VMWARE CORPORATION. http://www.vmware.com.

[38] WHITAKER, A., SHAW, M., AND GRIBBLE, S. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)* (December 2002).

[39] WOLSKI, R., SPRING, N., AND HAYES, J. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99* (August 1999), IEEE, pp. 105–112. Earlier version available as UCSD Technical Report Number CS98-602.

[40] WOLSKI, R., SPRING, N. T., AND HAYES, J. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems* (1999). To appear. A version is also available as UC-San Diego technical report number TR-CS98-599.